

Triggers over XML Views of Relational Data

Feng Shao Antal Novak Jayavel Shanmugasundaram
Cornell University
{fshao, afn, jai}@cs.cornell.edu

Abstract

Current systems that publish relational data as XML views are passive in the sense that they can only respond to user-initiated queries over the XML views. In this paper, we propose an active system whereby users can place triggers on (unmaterialized) XML views of relational data. In this architecture, we present scalable and efficient techniques for processing triggers over XML views by leveraging existing support for SQL triggers in commercial relational databases. We have implemented our proposed techniques in the context of the Quark system built on top of IBM DB2. Our performance results indicate that our proposed techniques are a feasible approach to supporting triggers over XML views of relational data.

1. Introduction

XML has emerged as a dominant standard for information exchange on the Internet. However, a large fraction of data continues to be stored in relational databases. Consequently, there has been a lot of interest in publishing relational data as XML. A powerful and flexible way to achieve this goal is to create XML views of relational data [11, 20, 23, 29]. In this way, the data can continue to reside in relational databases, while Internet applications can access the same data in XML format through the XML view. This architecture is shown in Figure 1. As a concrete example, consider a supplier that stores its product catalog information in a relational database. In order to expose the product catalog as an XML web service to buyers, the supplier can create an XML view of the product catalog and expose this view as a web service.

Current systems that support XML views of relational data are *passive* in the sense that they can only support user-initiated queries over the views. For instance, in the web services example above, current systems only allow buyers to explicitly initiate a request to query the catalog for products of interest. In this paper, we propose an *active* system that allows users to specify triggers over XML views. Thus,

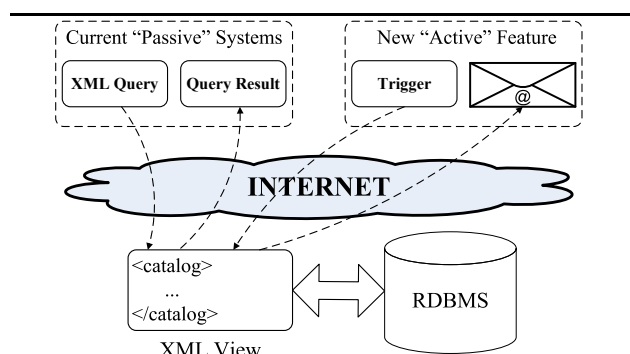


Figure 1. XML views of relational data.

a buyer can set a trigger to be notified whenever a new product is introduced, or when a product of interest goes out of stock, without having to repeatedly query the XML view to detect these changes.

At a high level, there are two approaches to supporting triggers over XML views. The first approach is to materialize the entire XML view, store it in an XML database, and implement XML triggers in this database. However, this approach suffers from the overhead of replicating and incrementally maintaining the materialized XML view on every relational update that affects the view, even though users may only be interested in relatively rare events. Another practical downside of this approach is that it requires a full-function XML database that supports incremental updates and triggers, even though the underlying relational database supports all of this functionality and is typically much more optimized for these tasks. Therefore, in this paper, we propose the alternative approach of translating XML triggers into *SQL triggers* over the relational data. The primary benefits of this approach are that it fully leverages sophisticated relational technology, does not require an XML database, and avoids having to materialize the XML view.

The main technical contribution of this paper is a systematic way to translate triggers over XML views of relational data into SQL triggers. This translation is fairly challenging because XML triggers can be specified over complex nested XML views with nested predicates, while SQL triggers can only be specified over flat relational tables. Consequently, even *identifying* the parts of an XML view that could have

changed due to a (possibly deeply nested) SQL update is a non-trivial task, as is the problem of *computing* the old and new values of an updated fragment of the view. Another issue is that current commercial relational databases are not very scalable with respect to the number of SQL triggers even though we expect a large number of XML triggers to be specified over XML views exposed as web services.

In this paper, we address the above challenges. Specifically, our two main contributions are: (1) a system architecture for supporting triggers over XML views of relational data (Section 3), and (2) an algorithm for identifying and computing changes in an XML view based on possibly deeply nested relational updates (Section 4). We also show how prior work on scalable trigger processing [14, 5] can be adapted for the XML view problem (Section 5).

We have implemented our proposed techniques in the context of the Quark system built over IBM DB2. One of the original goals of Quark (like XPERANTO [23] and SilkRoute [11]) was to support queries over XML views of relational data. By integrating with Quark, we were able to leverage many of the techniques originally developed for querying XML views, and adapt them to the trigger problem. This suggests that our techniques can be easily integrated into systems that already support queries over XML views of relational data (including relational database systems with built-in XML publishing support). Our performance results using our prototype show that our proposed techniques provide an efficient and scalable way to support triggers over XML views of relational data.

While our focus is on triggers over XML views, our techniques also apply to the less general problem of triggers over (flat) relational views. We note that current relational systems only support *INSTEAD OF* triggers [24] over unmaterialized views. Using *INSTEAD OF* triggers, users can manually specify how *updates on a view* are to be translated into updates on base tables. In contrast, we are solving the problem of automatically inferring when *updates on base tables* cause triggers on a view to be fired. We are not aware of any published work or commercial systems that support such SQL triggers over unmaterialized views.

2. Background

We have developed our trigger processing techniques in the context of the Quark system, which is similar to XPERANTO [23] in its support for querying XML views. We thus present an overview of XPERANTO, and also provide some background on XML and SQL triggers. We note that although our techniques are implemented in Quark, they are applicable to any XML publishing system.

Operator	Description
Table	Represents a relational table
Project	Computes results based on its input
Select	Restricts its input
Join	Joins two or more inputs
Groupby	Applies aggregate functions and grouping
Union	Unions inputs and removes duplicates
Unnest	Applies super-scalar functions to input

Table 1. XQGM operators.

2.1. XPERANTO Overview

In order to publish relational data as XML, XPERANTO first automatically creates a default view of the relational data. The default view, which is not materialized, is a simple mapping from relational tables to XML elements. Users can create their own application-specific views by specifying the transformation from the default view using XQuery. As an example, consider a relational database and its default view shown in Figure 2 (the database contains products and vendors for each product; primary keys are capitalized). Now suppose this database is exposed as a (virtual) XML view in which vendors are nested under products, with the restriction that only products sold by *at least two* vendors appear in the view. The XQuery view definition corresponding to this view is shown in Figure 3, and it is materialized in Figure 4.

While there are many details about query processing in XPERANTO that are not relevant here, one important relevant aspect is XQGM (the XML Query Graph Model). XQGM is used to represent and manipulate XQuery queries and views. XQGM consists of a set of operators and functions. The set of operators is shown in Table 1. Each operator produces a set of output tuples whose column values are XML nodes/values. Various functions can be embedded in operators to represent the manipulation of XML nodes.

As an illustration, the XQGM graph for the view definition in Figure 3 is shown in Figure 5. Operators (boxes) 1 and 2 produce the tuples in the *product* and *vendor* tables, respectively. Box 3 joins each vendor with the product it sells, and box 4 constructs a `<vendor>` element for each of these tuples. Box 5 then groups the elements by product name: the *aggXMLFrag()* function groups all the vendor elements in a group into a sequence, while the *count* function counts the number of vendors per group. Box 6 selects only the tuples with *count* ≥ 2 . Finally, boxes 7-9 create a `<product>` element for each product, group these into a single sequence, and produce a `<catalog>` element containing this sequence.

product		
PID	pname	mfr
P1	CRT 15	Samsung
P2	LCD 19	Samsung
P3	CRT 15	Viewsonic

vendor		
VID	PID	price
Amazon	P1	100.00
Bestbuy	P1	120.00
Circuitcity	P1	150.00
Buy.com	P2	200.00
Bestbuy	P2	180.00
Bestbuy	P3	120.00
Circuitcity	P3	140.00

```

<db>
  <product>
    <row><pid>P1</pid>
      <name>CRT 15</name>
      <mfr>Samsung</mfr>
    </row>
    ...
  </product>
  <vendor>
    <row><vid>Amazon </vid>
      <pid>P1</pid>
      <price>100.00</price>
    </row>
    ...
  </vendor>
</db>

```

Figure 2. Example database and its default view.

```

create view catalog as {
  <catalog>
    {for $sprodname in distinct(view("default")/
      product/row/pname)
      let $products := view("default")/product/
        row[./pname = $sprodname]
      let $vendors := view("default")/vendor/
        row[./pid = $products/pid]
      where count($vendors) >= 2
      return <product name={$sprodname}>
        { for $vendor in $vendors
          return <vendor>
            { $vendor/* }
          }
        }
    }
  </catalog> }

```

Figure 3. XML view definition.

```

<catalog>
  <product name="CRT 15">
    <vendor>
      <pid>P1</pid>
      <vid>Amazon</vid>
      <price>100.00</price>
    </vendor>
    <vendor>
      <pid>P1</pid>
      <vid>Bestbuy</vid>
      <price>120.00</price>
    </vendor>
    ...
  </product>
  <product name="LCD 19">
    ...
  </catalog>

```

Figure 4. Catalog view.

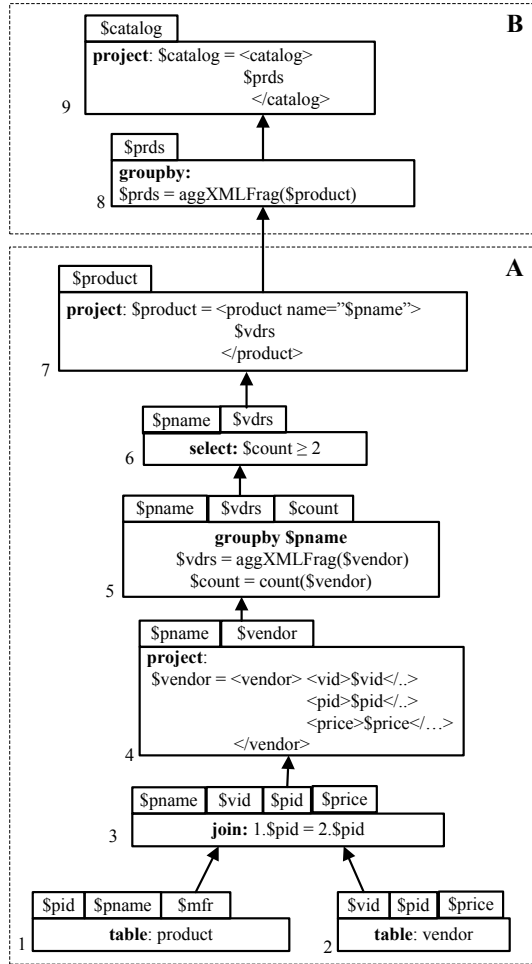


Figure 5. XQGM for the catalog view

2.2. XML Trigger Specification Language

We use a subset of the trigger specification language proposed by Bonifati et al. [2], whose syntax is shown below:

```

CREATE TRIGGER Name AFTER Event
ON Path WHERE Condition DO Action

```

A trigger has a unique *Name*. The *Event* specifies the

operation that activates the trigger, and can be either UPDATE, INSERT, or DELETE. *Path* is an XPath expression that specifies the portion of the XML view that is to be monitored for the event. *Condition* is a Boolean XQuery expression that specifies the condition under which the trigger is to be fired. When the condition is satisfied, the *Action* is performed; in our system, the action is a call to an external function which takes in XQuery expressions as parameters. Finally, two variables, OLD_NODE and NEW_NODE, are bound to the value of the node specified by *Path* before and after the *Event*; they may be referenced in the *Condition* and the *Action*. (When the *Event* is INSERT or DELETE, only the NEW_NODE or OLD_NODE, respectively, can be used.)

An example trigger over the view in Figure 3 is shown below. On any update to a product whose name was “CRT 15” (before the update), the trigger invokes an external function `notifySmith()` with the new value of that product. Note that the trigger will be fired not only for direct updates to a `<product>` element, but also for updates to its descendant nodes (i.e. vendors selling that product).

```

CREATE TRIGGER Notify AFTER Update
ON view('catalog')/product
WHERE OLD_NODE/@name = 'CRT 15'
DO notifySmith(NEW_NODE)

```

2.3. SQL Triggers

In contrast to XML triggers, which are specified on XML nodes, SQL triggers [6, 7] are fired when an event (INSERT, UPDATE, or DELETE) occurs on a specific relational table. When an SQL trigger is activated, it has access to the before-update and after-update versions of the affected rows through *transitional tables*. We use the notation $\nabla table$ to denote the transitional table that contains the updated rows before an update, and $\Delta table$ to denote the transitional table that contains the updated rows after an update ($\nabla table$ is empty for INSERT triggers, and $\Delta table$ is empty for DELETE triggers). For example, if product P1 goes on sale at Amazon, then the transitional tables might look like:

$\nabla vendor$			$\triangle vendor$		
<i>vid</i>	<i>pid</i>	<i>price</i>	<i>vid</i>	<i>pid</i>	<i>price</i>
Amazon	P1	100.00	Amazon	P1	75.00

3. Semantics and System Architecture

We now formalize the semantics of triggers on views, and then present our system architecture.

3.1. Semantics of Triggers on XML Views

In order to define the semantics of triggers on views, we need a precise definition of when an XML element in a view is said to be updated, inserted, or deleted. This in turn requires us to define the *identity* of an element in the view (so that we can talk about that element being updated, inserted or deleted). Note that the issue of identity is not as problematic for triggers over native XML data because each physical XML element has a well-defined notion of identity based on the XML data model. In contrast, XML elements in views are *virtual* and do not have a standard notion of identity. We now present an intuitive definition of the identity of XML elements based on the semantic structure of a view (in terms of the view's XQGM graph). The main idea is to use the notion of keys of XQGM operators to define the identity of nodes.

Definition 1 (Keys of XQGM Operators). *Given an operator o in XQGM graph G , a key of o is a minimal set of (existing or derivable) columns of o whose values uniquely identify each output tuple produced by o .*

As an illustration, a key of the table operator in box 1 in Figure 3 is the *pid* column (which is the *product* table's relational primary key). A key of the project operator in box 7 is the column containing the *\$pname* values (since the operator produces an output for each unique *\$pname*). Note that this key column is not directly present in the project operator but can be derived from its input operator.

In general, an operator can have more than one key. For instance, a relational table can have one column be a primary key and have a unique constraint on a different column. In such cases, we pick one of the keys to be the *canonical key*. For the table operator, we choose the primary key. The canonical keys for the other XQGM operators can be defined in terms of the canonical keys of its input operators (See Appendix A). For a tuple t produced by an operator o , we use the notation $v(t)$ to denote the value of (all columns of) t . We write $ckv_o(t)$ to denote the value of the canonical key columns of o for the tuple t . We denote the *top* operator of an XQGM graph G , which produces the final result of the graph, as o_G .

In order to define updates, inserts, and deletes on a view, we first formalize the notation for a *database transition*,

which is the result of UPDATES, INSERTS, and/or DELETES on relational tables. We do so in terms of the *database state*, where the database is in a state D before the transition, and a different state D' after the transition; we write the transition itself as $D \xrightarrow{*} D'$. When considering the effect of UPDATES, INSERTS, and/or DELETES to a single table T (as is the case when a SQL trigger on T is fired), we denote the transition as $D \xrightarrow{T} D'$. The result of evaluating operator o in state D is written $R(o, D)$.

We now define updates, inserts and deletes on views.

Definition 2 (View Trigger Updates). *A tuple t is said to be updated in view G by relational transition $D \xrightarrow{*} D'$ iff $t \in R(o_G, D)$, and $\exists t'(t' \in R(o_G, D') \wedge ckv_{o_G}(t) = ckv_{o_G}(t') \wedge v(t) \neq v(t'))$.*

Definition 3 (View Trigger Inserts (Deletes)). *A tuple t is said to be inserted (deleted) in view G by relational transition $D \xrightarrow{*} D'$ ($D' \xrightarrow{*} D$) iff $t \in R(o_G, D')$, and $\neg \exists t'(t' \in R(o_G, D) \wedge ckv_{o_G}(t) = ckv_{o_G}(t'))$.*

Given the above definition of events, we use the semantics of XML triggers specified by Bonifati et al. [2]. Note that our events are well-defined only for operators with (canonical) keys. We thus need to define a class of views for which triggers are well-defined.

Definition 4 (Trigger-Specifiable Views). *A view with XQGM graph G is trigger-specifiable iff every operator in G has a (canonical) key.*

We require every operator (not just the top operator) in the view to have a canonical key because the user can specify a trigger on a nested element (and not just a top level element). We can prove the following theorem (See Appendix B).

Theorem 1: A view G is trigger-specifiable if all the table operators in G have (canonical) keys.

Thus, arbitrarily complex views can have triggers specified on them, so long as the underlying relational tables have primary keys (which is the common case). We can also relax this restriction for a certain class of views and triggers but we do not describe these relaxations here.

3.2. System Architecture

Our system architecture is shown in Figure 6. Users can create triggers (using the syntax in 2.2) on trigger-specifiable views. The *Path*, *Condition* and *Action* of the trigger are converted into their respective XQGM graphs (recall that *Path*, *Condition* and *Action* are all XPath or XQuery expressions, and hence can be converted to XQGM). The trigger *Event* and the *Path* graph are then analyzed by the Event Pushdown module to determine the minimal set of base relations on which inserts, updates, or deletes could cause the trigger to be fired.

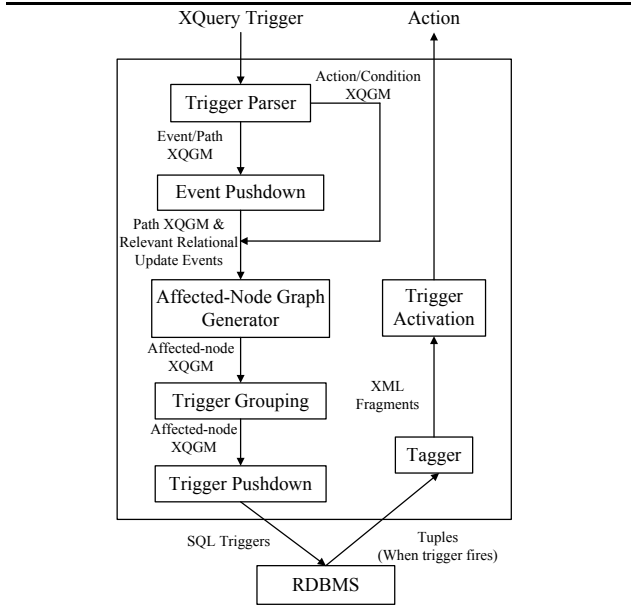


Figure 6. System architecture.

For each of these tables, the Affected-Node Graph Generator constructs an XQGM graph which, when evaluated, produces the `OLD_NODE` and `NEW_NODE` values for each affected XML node. This graph is then fed into the Trigger Grouping module, which groups similar triggers together for improved scalability. The Trigger Pushdown module takes the grouped trigger graph, pushes down selection conditions, and produces a set of SQL triggers, one for each relational event.

When activated, an SQL trigger issues a single SQL query to retrieve the relational data required for the actions of the XML triggers. The constant-space Tagger [23] then converts these results to XML. Finally, the Trigger Activation module activates the appropriate XML triggers and passes in the XML results as parameters to their actions.

In our implementation, we support a powerful subset of XQuery. Specifically, we support arbitrarily complex nested views with `FLWOR` expressions, quantified expressions, XPath expressions with `child/descendant` axes, arithmetic operators, comparison operators, and element constructors. We do not support XQuery type expressions or `sibling / parent / ancestor` XPath axes. For XML triggers, the *Path*, *Condition* and parameters to the *Action* can also be arbitrarily complex XQuery expressions with the same restrictions as for XML views. The grammar of supported expressions is specified in Appendix D. We note that our restrictions on XQuery expressions are an artifact of our current implementation and not an inherent limitation of our architecture. Also, while our system is implemented as middleware on top of a relational database, it can also be integrated into a database with XML publishing support.

Finally, a limitation of our current implementation is that

XML trigger(s) are fired for each SQL `INSERT`, `UPDATE`, or `DELETE statement`, rather than for each SQL *transaction* (which could contain more than one statement). This is not a limitation of our approach itself, but due to the fact that most commercial databases do not support SQL triggers at the transaction level; they only support SQL triggers at the granularity of a statement within a transaction. We note that our approach is general enough to support transaction level XML triggers if the underlying relational databases exposes transaction level SQL triggers.

3.3. Trigger Parsing and Event Pushdown

The first step in our architecture is to convert the trigger *Path*, *Condition* and *Action* XQuery expressions into XQGM; this is done in a manner similar to converting XQuery views to XQGM (see Section 2.1). In addition, we apply view composition rules [23] on the *Path* expression to identify the specific part of the view that the trigger monitors. For example, the trigger in Section 2.2 monitors the path `view('catalog')/product`. On composing this path with the catalog view, it produces the XQGM graph in Figure 5A. Note that this graph only produces products and not the entire catalog (since the trigger only monitors products).

The next step is to determine which events on which relational tables can cause the event specified in the XML trigger. This is similar to the problem of identifying events on the base tables that can affect materialized views [4] and violate constraints [3]. We adopt a similar approach to identifying relevant events on base tables and present the details in Appendix C. In our example, we are interested in `UPDATE` on the result of Box 7 in Figure 5A; this can be caused either by an `UPDATE` on the *product* table, or by an `INSERT`, `UPDATE` or `DELETE` on the *vendor* table.

4. Affected-Node Graph Generation

The goal of the Affected-Node Graph Generation module (see Figure 6) is to produce XQGM graphs that compute the input parameters for the trigger action. Specifically, the module takes as input the XQGM graphs for the *Path*, *Condition*, and parameters for the *Action*, and also the set of relational tables identified by the Event Pushdown module. For *each* of these tables, it produces an XQGM graph that computes the transformation from the relational transition tables to the parameters for the trigger action.

Our high-level approach is to produce a single XQGM graph, G_{params} , consisting of three parts, as shown in Figure 7. $G_{affected}$ produces a `(OLD_NODE, NEW_NODE)` tuple for each affected node of the view. G_{cond} , the XQGM graph corresponding to the *Condition*, evaluates the condition predicate which is then used to filter out any tuples that do not satisfy the condition. G_{action} then computes the

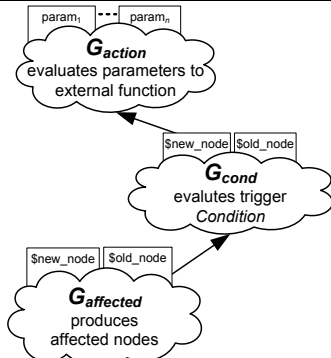


Figure 7. G_{params} : Producing parameters to *Action*.

XQuery expressions given as parameters to the *Action*. The main technical contribution of this section is an algorithm to produce $G_{affected}$.

4.1. Technical Challenges in Producing $G_{affected}$

On the surface, the problem of producing $G_{affected}$ may appear similar to the incremental view maintenance problem (where the goal is to compute changes to a materialized view based on updates to the base data). However, there are three new challenges that arise in our context, which require the development of new techniques.

First, as mentioned in the introduction, one of our design goals is to *not* materialize the XML view. We avoid materialization because (a) it would require an additional sophisticated XML database that can support incremental view updates, and (b) it would require the view to be updated for *every* relevant relational update even though user triggers may have very selective predicates¹. In contrast, most incremental view maintenance algorithms (e.g., [1, 4, 9, 10, 12, 15, 16, 21]) assume that the view is materialized, and use the materialized old value of a data item to compute its new value. We thus need to devise techniques that can directly compute the relevant new values from the base data.

Second, in producing $G_{affected}$, we need to compute new and old values after an update. In contrast, only the new value needs to be computed for materialized views. Thus, even materialized view techniques that can compute new values without using materialized old values (e.g., [18, 19]) are not directly applicable because they cannot compute (old value, new value) pairs. This problem is especially acute for INSERT/DELETE events because they introduce specific restrictions on whether the old/new values can appear in the view before/after an update (Definition 3).

¹ Note that if we chose to materialize the view, *all* items in the view (even those that do not satisfy any trigger selection predicate) would have to be incrementally maintained, because any item could become the *old value* of an updated item that *does* satisfy a trigger predicate.

Finally, the third (and perhaps most important) challenge arises due to nested predicates in XQuery. For instance, in Figure 5, we have multiple group-by (nesting) operators along with a selection predicate on a group-by aggregate value. While prior work on view maintenance for object-oriented [12, 16, 21], nested relational [15] and semi-structured [1, 9, 10] databases support nesting, they do not work with nested predicates. To understand why, consider the following example where a transaction inserts a row into the *vendor* table. The corresponding transition table is:

$\Delta vendor$		
<i>vid</i>	<i>pid</i>	<i>price</i>
Amazon	P2	500.00

Intuitively, for the XML view in Figure 5A, the above insert corresponds to an update of the “LCD 19” XML product (since a new vendor is added to this product). However, it turns out that the change computation technique (also referred to as the *propagate phase* [18]) commonly used for view maintenance will not detect this update. Specifically, most view maintenance algorithms compute changes to a view by replacing an updated table in the view definition with its corresponding transition table. In our example, this corresponds to replacing the *vendor* table in Figure 5A with the $\Delta vendor$ table, and evaluating the resulting query to compute the changes to the view. However, since $\Delta vendor$ has a single row, boxes 2, 3 and 4 will each produce a single row and the selection in box 6 will return no rows since $\$count = 1$. Hence, no changes will be detected!

As the reader has probably observed, the above problem arises because we are trying to compute changes for nested predicate views using only tuples from the transition table. This results in inaccurate aggregate values and hence misses some relevant updates (it can also introduce spurious updates in other cases). We thus need to devise techniques for correctly computing changes for views with nested predicates. We note that [1] does present a technique for computing changes to views with existential predicates and a single level of nesting (existential predicates can be viewed as a very specific form of a select over an aggregation). However, we are not aware of any prior technique that can handle complex query predicates at arbitrary levels of nesting.

4.2. Proposed Algorithm

We now present our algorithm for producing $G_{affected}$. The algorithm first detects the keys of the XML nodes affected by an update (*affected keys*) and then use the affected keys to compute the actual node *values*. Our main contributions are (a) a technique for correctly determining affected keys even when the view has arbitrary nested predicates, and (b) a technique for using the affected keys to generate (OLD.NODE, NEW.NODE) pairs that satisfy the definition of trigger events, without using any materialized data.

In what follows, we use the following notations. G is the original *Path* graph; B is the post-update version of the table in question (keep in mind that this algorithm is invoked once for *each* base relational table); B_{old} is the pre-update version of this table; G_{old} is a graph identical to G with the sole exception that B is replaced by B_{old} . While most DBMSes do not expose the B_{old} table directly, it can easily be constructed using a query of the form [7]: (SELECT * FROM B) EXCEPT (SELECT * FROM ΔB) UNION (SELECT * FROM ∇B).

4.2.1. CreateAKGraph: Finding Affected Keys. Figure 8 presents our algorithm for determining the affected keys. The algorithm takes as input an operator O (the top operator in the *Path* graph), a base table T , and a transitional table dT (which is either ΔT or ∇T). It returns a new operator, O' , which is the top operator of an XQGM graph such that $O \bowtie O'$ will produce exactly the subset of O 's output tuples which are affected by the relational update captured by dT .

In order to determine the keys of G affected by ΔB (or ∇B), we traverse G (or G_{old} , respectively) in depth-first order, building up a parallel graph $G_{\Delta key}$ (or $G_{\nabla key}$, respectively). At each step, we maintain the following invariant: for each operator o in G and the corresponding operator o_{Δ} in $G_{\Delta key}$, joining o and o_{Δ} on the key of o will produce exactly those tuples from the result of o that were affected by ΔB . Thus, if o is the top operator of G , then the corresponding o_{Δ} operator provides a way to identify the nodes in the result of G that are affected by relational update.

We now walk through the algorithm using the *Path* graph in Figure 5A, for the case of an UPDATE on *vendor* (the other cases are similar). At the leaf level, a **Table**($\Delta vendor$) operator will first be created. Clearly the invariant holds at this point: joining **Table**($\Delta vendor$) with **Table**(*vendor*) on the \$vid column (the key of **Table**(*vendor*)) would produce exactly the *vendor* tuples that changed. The result after this step is shown in Figure 9.

Box 3 (the **Join** operator) merely propagates the \$vid column without creating any new operators in $G_{\Delta key}$. The corresponding operator in $G_{\Delta key}$ remains **Table**($\Delta vendor$), and the invariant still holds: joining box 3 with $\Delta vendor$ on \$vid would produce exactly those product-vendor pairs affected by $\Delta vendor$. For box 4, we can similarly just propagate the \$vid column without having to create any new operators in $G_{\Delta key}$.

We then arrive at box 5, a **GroupBy** operator. Since a **GroupBy** operator aggregates multiple input values, any update to any one the input values in a group can change the aggregate result for that group. We therefore need a way to create an operator o_{Δ} in $G_{\Delta key}$ that only produces the keys of those groups affected by the update. First, we join the operator *below* the current **GroupBy** (box 4) with its

```

1: CreateAKGraph ( $O, T, dT$ ) : ( $Operator, Key$ )
   { $O$  is an operator;  $T$  and  $dT$  are table names.}
2:    $I \leftarrow$  input operators to  $O$ 
3:   if  $O.type = \text{Table}$  then
4:     if  $O.tableName = T$  then
5:        $PK \leftarrow$  primary key of table  $T$ 
6:        $(O', K) \leftarrow (\text{Project}_{(PK)}(\text{Table}(dT)), PK)$ 
7:     else
8:        $(O', K) \leftarrow (\emptyset, \emptyset)$ 
9:     end if
10:  else if  $O.type = \text{GroupBy}$  then
11:     $I' \leftarrow \text{CreateAKGraph}(I, T, dT)$ 
12:    if  $I' = \emptyset$  then
13:       $(O', K) \leftarrow (\emptyset, \emptyset)$ 
14:    else
15:       $J \leftarrow \text{Join}_{(key(I'))}(I, I')$ 
16:       $K \leftarrow$  grouping columns of  $O$ 
17:       $O' \leftarrow$  new GroupBy on  $J$  with grouping cols  $K$ 
18:    end if
19:  else if  $O.type = \text{Select}$  or  $O.type = \text{Project}$  then
20:     $(I', key_I) \leftarrow \text{CreateAKGraph}(I, T, dT)$ 
21:     $(O', K) \leftarrow (I', key_I)$ 
22:  else if  $O.type = \text{Join}$  then
23:    {Assuming, without loss of generality, that  $|I| = 2$ .}
24:    for all  $i \in I$  do
25:       $(i', k) \leftarrow \text{CreateAKGraph}(i, T, dT)$ 
26:      if  $i' \neq \emptyset$  then
27:         $I' \leftarrow I' \cup \{i'\}$ 
28:         $K \leftarrow K \cup \{k\}$ 
29:      end if
30:    end for
31:    if  $|I'| = 0$  then
32:       $O' \leftarrow \emptyset$ 
33:    else if  $|I'| = 1$  then
34:       $O' \leftarrow I'$ 
35:    else
36:      {Create a union of cross-products}
37:       $J_a \leftarrow \text{Project}_{(K)}(\text{Join}(I'_0, I'_1))$ 
38:       $J_b \leftarrow \text{Project}_{(K)}(\text{Join}(I'_0, I'_1))$ 
39:       $O' \leftarrow \text{Union}(J_a, J_b)$ 
40:    end if
41:  else if  $O.type = \text{Union}$  then
42:    {Assuming, without loss of generality, that  $|I| = 2$ .}
43:    for all  $i \in I$  do
44:       $(i', k) \leftarrow \text{CreateAKGraph}(i, T, dT)$ 
45:      if  $i' \neq \emptyset$  then
46:         $I' \leftarrow I' \cup i'$ 
47:         $K_i \leftarrow k$ 
48:      end if
49:    end for
50:    {Construct the key, as described in Appendix A}
51:     $K \leftarrow \bigcup_{i \in I} \bigcup_{c \in K_i} M(c)$ 
52:    {Create union of all operators in the set  $I'$ }
53:     $O' \leftarrow \text{Union}(I')$ 
54:  end if
55:  {Ensure that  $O$ 's key is propagated}
56:  Add  $K$  to  $O.outputColumns$ 
57:
58:  Return ( $O', K$ )
59:

```

Figure 8. Algorithm for producing affected keys.

corresponding operator in $G_{\Delta key}$ (1_{Δ}). By the algorithm invariant, we can infer that this new join produces exactly the set of input values to the **GroupBy** operator that have changed. Thus, to identify the *keys* of all affected groups, we simply need to project distinct values of the grouping columns, which we achieve by creating a new **GroupBy** operator (3_{Δ}) which groups on this column (\$pname). The $G_{\Delta key}$ graph at this point is shown in Figure 10.

The final two operators are **Select** and **Project** operators and, like boxes 3 and 4, merely propagate the key col-

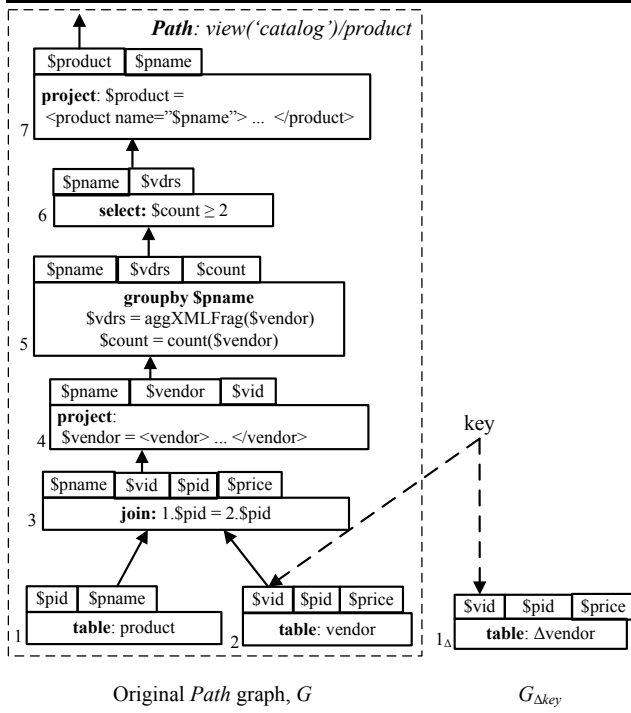


Figure 9. CreateAKGraph: Step 1.

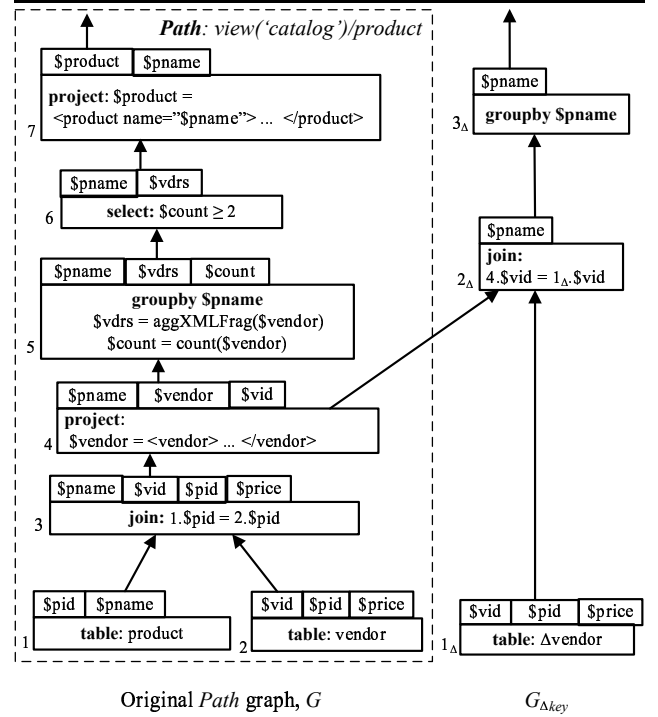


Figure 11. The complete $G_{\Delta key}$ graph.

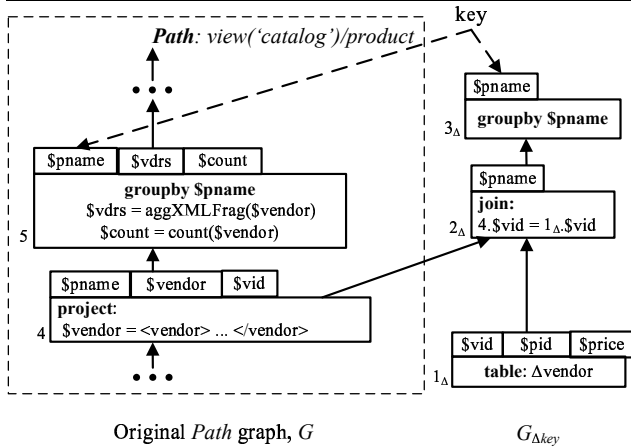


Figure 10. CreateAKGraph: GroupBy operator.

umn ($\$pname$); no new boxes are created in $G_{\Delta key}$. The final graph is shown in Figure 11. Note that the only difference from Figure 10 is that box 7 now propagates $\$pname$.

The application of **GetAKGraph** is virtually identical for producing $G_{\nabla key}$, so we don't walk through it here; the only difference is that $vendor_{old}$ and $\nabla vendor$ are used instead of $vendor$ and $\Delta vendor$, respectively.

4.2.2. CreateANGraph: Producing Affected Nodes.

At this stage, for a given relational table-event pair, we have the *Path* graphs (G and G_{old}) and the affected-key graphs ($G_{\Delta key}$ and $G_{\nabla key}$). Our goal now is to produce the $G_{affected}$ graph, which produces the (OLD_NODE,

NEW_NODE) pairs corresponding to the relational update. The algorithm for producing $G_{affected}$ is given in Figure 12. We begin by creating a **Union** operator merging the $G_{\Delta key}$ and $G_{\nabla key}$ subgraphs. This captures the notion that we need to compute the (OLD_NODE, NEW_NODE) pairs for nodes affected by *either* insertions or deletions at the relational level. Next, the result of the union is joined with G to get NEW_NODE, and joined with G_{old} to get OLD_NODE. Finally, OLD_NODE and NEW_NODE are joined on the key. The *type* of this join depends on the XML trigger *Event*: an UPDATE has both OLD_NODE and NEW_NODE (hence an inner join), an INSERT has only NEW_NODE (hence a left anti join), and DELETE, only OLD_NODE (hence a right anti join). In our example, we perform an inner join since we are monitoring UPDATES. In some special cases of UPDATES (not relevant to our example), we need to explicitly check whether the values of OLD_NODE and NEW_NODE are different; we refer the reader to Appendix F for more details of these special cases and related optimizations. Figure 13 shows the final $G_{affected}$ graph for our example.

We can prove the following correctness theorem for UPDATES (and similarly for INSERTS and DELETES) (See Appendix E).

Theorem 2: Given an UPDATE event, view graph G , and table T , **CreateANGraph** produces graph $G_{affected}$ such that for all valid database transitions $D \xrightarrow{T} D'$,


```

1: CreateANGraph
   ( $E : \text{Event}, G : \text{XQGMGraph}, B : \text{String}$ ) :
   {Build up the affected-key graph for  $\Delta B$ , and project out just the
   key column(s)}
2:  $O_{\Delta key} \leftarrow \text{CreateAKGraph}(o_G, B, \Delta B)$ 
3:  $G_{\Delta key} \leftarrow \text{Project}_{(O_{\Delta key}, key)}(O_{\Delta key})$ 
   {Do the same for  $\nabla B$ }
4:  $O_{\nabla key} \leftarrow \text{CreateAKGraph}(o_{G_{old}}, B_{old}, \nabla B)$ 
5:  $G_{\nabla key} \leftarrow \text{Project}_{(O_{\nabla key}, key)}(O_{\nabla key})$ 
   {Take the union of the affected keys}
6:  $O_u \leftarrow \text{Union}(G_{\Delta key}, G_{\nabla key})$ 
   {And join it back with  $G/G_{old}$  to produce NEW_NODE/OLD_NODE}
7:  $O_{new} \leftarrow \text{Join}_{(O_u, key=G_{key})}(O_u, G)$ 
8:  $O_{old} \leftarrow \text{Join}_{(O_u, key=G_{old, key})}(O_u, G_{old})$ 
   {Finally, the way we produce  $G_{affected}$  depends on the type of event}
9: if  $E = \text{UPDATE}$  then
9:   {Inner join; we want those nodes which are present in both
   OLD_NODE and NEW_NODE}
10:  $G_{affected} \leftarrow \text{Join}_{(key)}(O_{new}, O_{old})$ 
11: If required,  $G_{affected} \leftarrow \text{Select}_{(OLD\_NODE \neq NEW\_NODE)}(G_{affected})$ 
12: else if  $E = \text{INSERT}$  then
12:   {Here we only want those nodes which are not present in
   OLD_NODE}
13:  $G_{affected} \leftarrow \text{LeftAntiJoin}_{(key)}(O_{new}, O_{old})$ 
14: else if  $E = \text{DELETE}$  then
15:  $G_{affected} \leftarrow \text{RightAntiJoin}_{(key)}(O_{new}, O_{old})$ 
16: end if

```

Figure 12. Algorithm for producing $G_{affected}$.

$(OLD_NODE, NEW_NODE) \in R(o_{G_{affected}}, D')$ iff $OLD_NODE \in R(o_G, D) \wedge NEW_NODE \in R(o_G, D') \wedge ckv_{o_G}(OLD_NODE) = ckv_{o_G}(NEW_NODE) \wedge v(OLD_NODE) \neq v(NEW_NODE)$.

4.3. Adding Condition and Action

Finally, as described in the beginning of this section, we need to produce G_{params} , the graph that produces parameters for the *Action* after selecting only the (OLD_NODE, NEW_NODE) pairs that satisfy the trigger condition (see Figure 7). G_{params} is produced by converting the XQuery expressions for *Condition* and parameters of *Action* into XQGM (to produce G_{cond} and G_{action} , respectively), and stacking these graphs on top of $G_{affected}$. Figure 13 shows G_{params} for our running example.

5. Trigger Grouping and Pushdown

Given G_{params} for each relevant table-event pair, the final two steps in generating SQL triggers are Trigger Grouping and Trigger Pushdown (Figure 6). We describe each in turn.

5.1. Trigger Grouping

A simple approach to producing SQL triggers is to create one SQL trigger for each G_{params} graph of an XML trigger. However, this approach will not be very efficient because the number of SQL triggers produced will be at least as many as the number of XML triggers (which we expect

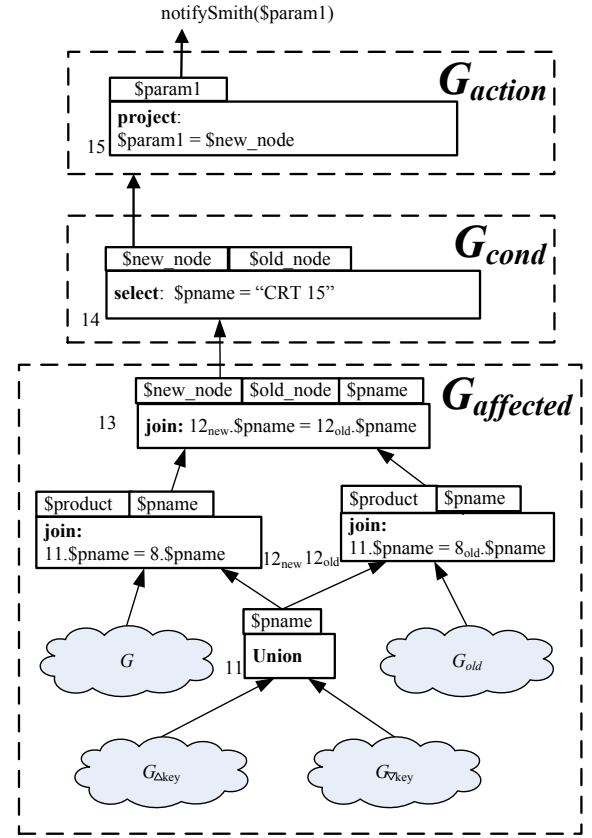


Figure 13. The final G_{params} graph.

to be large), and current relational database systems are not very scalable with respect to the number of SQL triggers. We thus explore techniques for grouping structurally similar G_{params} graphs together, and producing a single SQL trigger for each group. We note that our focus is not on developing new techniques for grouping triggers; rather, our focus is on adapting existing techniques [5, 14] to work with nested XML views.

For the purposes of this paper, we only consider grouping G_{params} graphs that differ in the constant value(s) of a selection condition (this corresponds to grouping structurally similar XML triggers that only differ in selection constant(s) in the WHERE clause). For instance, we would consider grouping the G_{params} graph in Figure 13 with another graph that is identical in all respects but for the constant value of the selection condition in box 14 (which could, say, select “LCD 19” instead). The proposed approach can also be extended for grouping joins, but we do not discuss this extension here.

The first step is to create a *constants* table [14] for each group of structurally similar G_{params} graphs. The *constants* table has a *TrigIDs* column, which identifies the XML triggers which share a particular set of constants, followed by as many columns as there are constants in the triggers. For instance, if in our example, XML triggers 1 and 2 both share

the value CRT 15, while trigger 3 uses LCD 19, the *constants* table would look like:

TrigIDs	ConstI
1,2	CRT 15
3	LCD 19

Given the *constants* table, the standard grouping technique [5, 14] is to directly convert the selection condition with constant(s) into a join with the *constants* table, as shown in Figure 14 for our example. In this way, multiple individual selections are converted into a single join, and are hence more efficient.

However, this direct replacement of a select with a join does not work for complex nested XML conditions. To see why, let us assume that the WHERE condition in our example is modified to be of the form: **count**(NEW_NODE/vendor[./price < x]) $\geq y$ (i.e., the new node contains at least y vendors who sell an item for less than x ; here x and y can be different constants for different XML triggers). In this case, the condition contains a selection (./price < x) nested under a group-by (**count**) nested under another selection ($\geq y$). Simply replacing a selection (such as ./price < x) with a corresponding join would be incorrect because this would change the output cardinality of the operator (due to the join with the *constants* table, where multiple triggers could be fired); this would in turn change the group-by (**count**) result, thereby producing wrong results.

To address this issue, we propose a simple yet powerful approach that works for arbitrarily complex nested selections. The basic idea is to use the *constants* table to set up a *correlation* in the G_{params} graph to produce a $G_{grouped}$ graph, as shown in Figure 15. Conceptually, this means that the G_{params} graph is evaluated once for each row in the *constants* table (i.e., for each unique set of constants). While this will certainly produce the correct results, it is likely to be inefficient because we still do selections one by one for each unique set of constants. However, the key idea now is to decorrelate this graph using query rewrite techniques developed for SQL [22] and XML [23] queries. Decorrelation converts correlated selections to joins [22, 23] (as we desire) and also preserves the correct semantics of the graph by adding appropriate grouping columns to group-by operators so that nested selections are handled correctly.

5.2. Trigger Pushdown

Given a decorrelated $G_{grouped}$ graph for a table-event pair, the next step is to generate a SQL trigger that, when activated, produces the output of $G_{grouped}$. In generating this SQL trigger, we leverage techniques developed for publishing relational data as XML. Specifically, we apply selection/join pushdown and tagger pull-up transformations [23] on $G_{grouped}$ to generate a single sorted outer union SQL

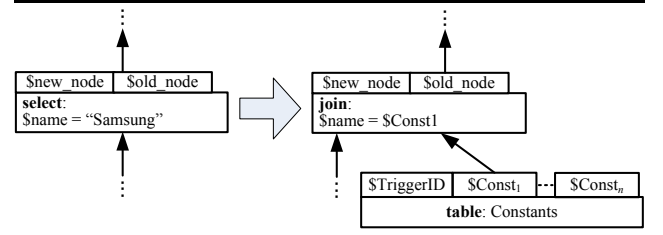


Figure 14. Converting select to join.

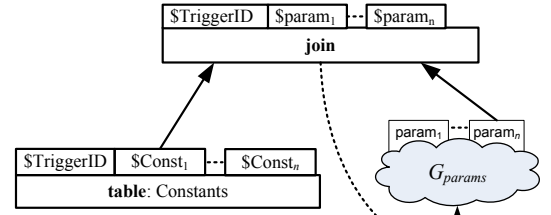


Figure 15. Correlated $G_{grouped}$ graph.

query whose results can be tagged in constant space to produce the XML output; this query becomes the body of the SQL trigger generated for the table-event pair.

We also employ an important optimization that is specific to triggers, which we now briefly describe. The main idea is to avoid directly computing distributive aggregate functions over B_{old} , the pre-update version of a table, whenever such functions appear in the view definition (such as a *count* over vendors, in our example). We perform this optimization for two reasons. First, when an aggregate function is specified in a view, we need to compute the aggregate for *both* B (the post-update version of the table) and B_{old} (the pre-update version of the same) since we need to produce both OLD_NODE and NEW_NODE values; computing an aggregate twice is likely to be expensive. Second, if we can avoid referencing B_{old} directly, we might be able to avoid materializing B_{old} (recall from Section 4.2 that B_{old} is not exposed directly and has to be explicitly computed).

We use the following technique to avoid directly computing an aggregate on B_{old} , while still producing the correct OLD_NODE values. We compute the result of the aggregation on B_{old} by using the result of the aggregation on B and the transition tables. Note that this approach is exactly the inverse of the incremental view maintenance problem (since we compute old aggregate values from new aggregate values)! Consequently, by switching the role of old values and new values, we can directly use existing incremental view maintenance techniques [19] to compute aggregates on B_{old} using just B and the transition tables.

The SQL trigger generated for our running example of an UPDATE on *vendor* is shown in Figure 16 (formatted to be more human-readable). The trigger first finds the affected keys by taking a union of the product names associated with tuples in the transition tables (lines 6-13). The trigger then

```

1 CREATE TRIGGER sqlTrigger1
2 AFTER UPDATE ON VENDOR
3 REFERENCING OLD_TABLE AS DELETED, NEW_TABLE AS INSERTED
4 FOR EACH STATEMENT
5
6 WITH AffectedKeys (name) AS (
7     SELECT P.name
8     FROM product AS P, INSERTED AS V
9     WHERE P.pid = V.pid
10 UNION
11     SELECT P.name
12     FROM product AS P, DELETED AS V
13     WHERE P.pid = V.pid),
14
15 ProductCount (name, numVendors) AS (
16     SELECT P.name, COUNT(*) AS numVendors
17     FROM AffectedKeys AS C, product AS P, vendor AS V
18     WHERE P.name = C.name
19     AND P.pid = V.pid
20     GROUP BY P.name),
21
22 MultiVendorProduct (name) AS (
23     SELECT name
24     FROM ProductCount
25     WHERE numVendors >= 2),
26
27 deltaCount (name, numVendors) AS (
28     SELECT P.name, 1
29     FROM product AS P, DELETED AS D
30     WHERE P.pid = D.pid
31 UNION ALL
32     SELECT P.name, -1
33     FROM product AS P, INSERTED AS I
34     WHERE P.pid = I.pid),
35
36 MultiVendorProduct_old (TrigIDs, name) AS (
37     SELECT C.TrigIDs, T2.name
38     FROM Constants1 AS C,
39     ( SELECT name
40       FROM ( SELECT name, numVendors
41             FROM ProductCount PC, Constants1 C
42             WHERE PC.name = C.Const1
43             UNION ALL
44             SELECT name, numVendors
45             FROM deltaCount DC, Constants1 C
46             WHERE DC.name = C.Const1
47           ) AS T(name, numVendors)
48       GROUP BY T.name
49       HAVING SUM(T.numVendors) >= 2
50     ) AS T2(name)
51     WHERE T2.name = C.name),
52
53 ProductInfo (pid, name, TrigIDs) AS (
54     SELECT P.pid, P.name, MVP_old.TrigIDs
55     FROM Product AS P, MultiVendorProduct AS MVP,
56     MultiVendorProduct_old AS MVP_old
57     WHERE MVP.name = MVP_old.name
58     AND MVP.name = P.name)
59
60 -- Produce the <product> elements
61 SELECT 1, PI.TrigIDs, PI.name, NULL, NULL
62 FROM ProductInfo AS PI
63
64 UNION ALL
65
66 -- Produce the <vendor> elements
67 SELECT 2, NULL, NULL, vid, price
68 FROM Vendor AS V, ProductInfo AS PI
69 WHERE V.pid = PI.pid
70
71 ORDER BY TrigIDs, pname, vid

```

Figure 16. The generated SQL trigger.

Parameter	Values (default in bold)
Hierarchy depth	2, 3, 4, 5
Number of leaf tuples ($\times 1000$)	32, 64, 128 , 256, 512, 1024
Leaf tuples per XML element	16, 32, 64 , 128, 256
Number of triggers	1, ..., 10,000 , ..., 100,000
Number of satisfied triggers	1 , 20, 40, 80, 100

Table 2. Experimental parameters.

computes the number of vendors for each affected product *after* the update (lines 15-20), and selects only those with more than one vendor as potential NEW_NODES (lines 22-25). Note that vendors are only computed for affected products by using regular query rewrite techniques to push down the join on affected keys [18, 23]. The trigger then computes the number of vendors for each affected product *before* the update by using the corresponding values *after* the update and the transition tables (lines 27-51); also note that the selection on the OLD_NODE name is transformed into a join due to trigger grouping. Finally, the action parameters are produced using a sorted outer union (lines 61-71).

6. Experimental Evaluation

We have developed and evaluated a prototype of the system architecture proposed in this paper. We observe that the *compile time* for an XML trigger, which is the time to manipulate the intermediate XQGM graphs and produce the final SQL trigger, is fairly small (a hundred milliseconds, even for a complex view), and is only expended once during the creation of the trigger. We therefore focus our evaluation on the *run time*, which is the overhead of evaluating the generated SQL trigger on an update to the underlying base table(s).

6.1. Experimental Setup

Our experimental setup is characterized by the parameters in Table 2. *Hierarchy depth* specifies the depth of the relational schema. For depth 2, we use the *product/vendor* schema and XML view described earlier. For deeper views, we add additional “ancestor” tables above *product*, so that each child table has a foreign key column referencing its parent’s primary key, and the XML view contains children nested inside of parents. *Number of leaf tuples* specifies the number of rows in the leaf (i.e., *vendor*) table—this is a measure of the size of the database. *Leaf tuples per XML element* is the number of leaf tuples contained in a top-level XML element produced by the view; this is a measure of the size of an individual XML element. The next two parameters specify the number of structurally similar XML triggers in the system, and the number of these triggers that are fired for each relational update, respectively. In all cases, the XML trigger was placed on the top-level XML element

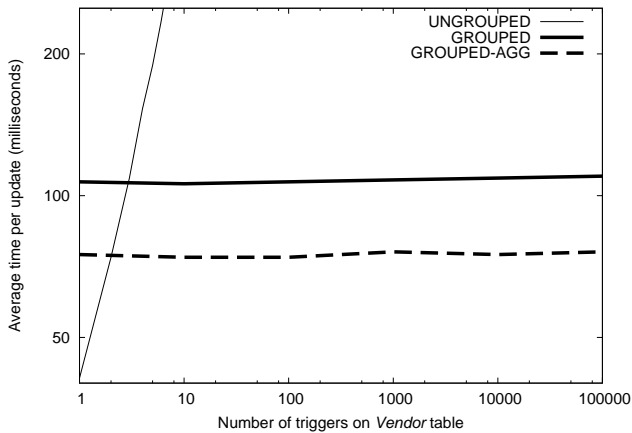


Figure 17. Varying the number of triggers.

in the view, and the $\text{count}(\dots) \geq 2$ predicate remained on the lowest level, that is, on the vendors. We defined the actions of the triggers to insert the entire `NEW_NODE` into a temporary table.

We evaluated three alternative implementations to evaluate the various aspects of our approach. `UNGROUPED` generates a different SQL trigger for each table-event pair corresponding an XML trigger. `GROUPED` extends `UNGROUPED` by implementing trigger grouping (Section 5.1). `GROUPED-AGG` extends `GROUPED` by optimizing aggregate computation on B_{old} (Section 5.2). Our experiments were performed on a Linux system with a 933MHz Pentium III processor and 1GB of main memory, running the DB2 8.1 commercial relational DBMS. We defined primary keys for all the relational tables and built appropriate indices on the key columns and other join columns. For each experiment, we varied one of the parameters in Table 2 and used default values for the rest (the default values are in bold). The run time was averaged over 100 independent updates to the *vendor* table, performed on a cold cache.

6.2. Varying Number of Triggers

Figure 17 shows the performance of the different approaches when we vary the number of triggers in the system. `UNGROUPED` does not scale well with the number of triggers because it does not benefit from shared computation across triggers. In contrast, `GROUPED` and `GROUPED-AGG` scale gracefully due to the grouping optimizations, with virtually no change in the update time as the number of XML triggers increases (note the log scale in the x-axis). This suggests that we can build a scalable XML trigger system over relational databases, even though relational triggers by themselves are not scalable. Finally, `GROUPED-AGG` provides about a 30% performance improvement over `GROUPED` due to the ag-

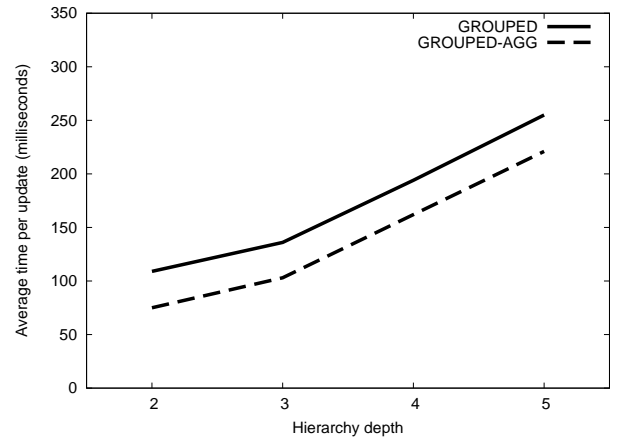


Figure 18. Varying the hierarchy depth.

gregation optimization.

6.3. Varying Hierarchy Depth

Figure 18 shows the effect of varying the depth of the XML view (we do not show `UNGROUPED` because of its bad scalability properties). `GROUPED` and `GROUPED-AGG` have the same relative performance and their run time increases approximately linearly with the depth. This is because, as the depth increases, the relational trigger must evaluate more joins to recreate the hierarchy. Further, the size of the produced result also increases because, although the number of leaf nodes remains constant, the number of intermediate nodes grows larger. The results indicate that we can get good performance for XML triggers even for deeply-nested XML views.

6.4. Summary of Other Results

We also varied the other parameters in Table 2 and we obtained similar results as in the previous two experiments: `GROUPED` and `GROUPED-AGG` scaled gracefully, and `GROUPED-AGG` provides at least a 30% improvement over `GROUPED`; this difference actually increases when we vary the number leaf tuples per XML element since the cost of aggregation increases. We refer the reader to Appendix G for further details.

7. Related work

There has been some recent work on supporting triggers in native XML databases [2, 17]. Our focus, however, is on implementing XML triggers using SQL triggers, without any need for a native XML database. A related class of systems is XML publish/subscribe (pub/sub) systems [5, 8, 13, 26, 17], where the goal is to efficiently match

streaming XML documents against a large set of subscriptions. However, most most pub/sub systems do not consider updates to XML documents, which is the main issue with XML triggers. Furthermore, even those that consider updates [17] work with native XML engines and do not exploit relational triggers. Our work is also related to the incremental maintenance of relational/nested-relational/object-oriented/semi-structured views; we refer the reader to Section 4.1 for the main technical differences with respect to view maintenance. In addition, our work also differs from XML view maintenance in that we exploit *relational* triggers and query processing.

8. Conclusion and Future Work

We have presented a systematic way of translating triggers over XML views of relational data into SQL triggers, and for translating relational updates into their corresponding XML updates. We have also presented experimental results that illustrate the feasibility and scalability of our approach. An interesting direction for future work is to see whether our general algorithm for detecting changes over complex XQuery views can be adapted for incrementally maintaining complex *materialized* XML views.

References

- [1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, J. Wiener, "Incremental Maintenance for Materialized Views over Semistructured Data", VLDB 1998.
- [2] A. Bonifati, D. Braga, A. Campi, S. Ceri, "Active XQuery", ICDE 2002.
- [3] S. Ceri, J. Widom, "Deriving Production Rules for Constraint Maintenance", VLDB 1990.
- [4] S. Ceri, J. Widom, "Deriving Production Rules for Incremental View Maintenance", VLDB 1991.
- [5] J. Chen, D. J. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", SIGMOD 2000.
- [6] R. Cochrane, K. Kulkarni, N. Mattos, "Active Database Features in SQL3", In N. Paton (ed.) *Active Rules in Database Systems*, Springer-Verlag, 1999.
- [7] R. Cochrane, H. Pirahesh, N. Mattos, "Integrating Triggers and Declarative Constraints in SQL Database Systems", VLDB 1996.
- [8] Y. Diao, P. Fischer, M. Franklin, R. To, "YFilter: Efficient and Scalable Filtering of XML Documents", ICDE 2002.
- [9] K. Dimitrova, M. El-Sayed, E. Rundensteiner, "Order-sensitive View Maintenance of Materialized XQuery Views", ER 2003.
- [10] M. El-Sayed, L. Wang, L. Ding, E. Rundensteiner, "An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views", WIDM 2002.
- [11] M. Fernandez, D. Suciu, W. Tan, "SilkRoute: Trading between Relations and XML", WWW 2000.
- [12] D. Gluche, T. Grust, C. Mainberger, M. Scholl, "Incremental Updates for Materialized OQL Views", DOOD 1997.
- [13] A. Gupta, D. Suciu, "Stream Processing of XPath Queries with Predicates", SIGMOD 2003.
- [14] E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, A. Vernon, "Scalable Trigger Processing", ICDE 1999.
- [15] A. Kawaguchi, D. Lieuwen, I. Mumick, K. Ross, "Implementing Incremental View Maintenance in Nested Data Models", DBPL 1997.
- [16] H. Kuno, E. Rundensteiner, "Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation", TKDE 10(5), 1996.
- [17] B. Nguyen, S. Abiteboul, G. Cobena, M. Preda, "Monitoring XML Data on the Web", SIGMOD 2001.
- [18] T. Palpanas, R. Sidle, R. Cochrane, H. Pirahesh, "Incremental Maintenance for Non-Distributive Aggregate Functions", VLDB 2002.
- [19] D. Quass, "Maintenance Expressions for Views with Aggregation", SIGMOD 1996.
- [20] M. Rys, "State-of-the-Art Support in RDBMS: Microsoft SQL Server's XML Features", Data Engineering Bulletin 24(2), 2001.
- [21] M. Rys, M. Norrie, H. Schek, "Intra-Transaction Parallelism in the Mapping of an Object-Model to a Relational Multi-Processor System", VLDB 1996.
- [22] P. Seshadri, H. Pirahesh, T. Leung, "Complex Query Decorrelation", ICDE 1996.
- [23] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk, "Querying XML views of relational data", VLDB 2001.
- [24] M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos, "On Rules, Procedures, Caching and Views in Data Base Systems", SIGMOD 1990.
- [25] D. Suciu, "Query Decomposition and View Maintenance for Query Languages for Unstructured Data", VLDB 1996.
- [26] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, J. Myllymaki, "Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems", SIGMOD 2004.
- [27] World Wide Web Consortium. "XQuery 1.0: An XML Query Language", W3C Working Draft, 12 Nov. 2003. See <http://www.w3.org/TR/2003/WD-xquery-20031112/>.
- [28] World Wide Web Consortium. "XQuery 1.0 and XPath 2.0 Functions and Operators", W3C Working Draft, 12 Nov. 2003. See <http://www.w3.org/TR/2003/WD-xpath-functions-20031112/>.
- [29] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, E. Rundensteiner, "Rainbow: mapping-driven XQuery processing system", SIGMOD 2002.

A. Definitions of Canonical Keys

Table 3 defines the canonical key for each XQGM operator (except for **Unnest**) in terms of the canonical keys of its input operator(s).

B. Proof of Theorem 1

Theorem 1. *A view of relational data, G , is trigger-specifiable if all the table operators in G have (canonical) keys.*

Proof. We need to prove that every operator in G has a canonical key. In Table 3, we define the canonical keys for every type of operator, except for **Unnest**, in terms of its input operator(s). Thus, if G does not contain any **Unnest** operators, then we can simply derive the canonical key for each operator o by applying the definitions in Table 3.

If G does contain **Unnest** operators, then it can be rewritten to an equivalent graph G' that does not contain any **Unnest** operators. This transformation is possible because G is an XML view of relational data and the underlying relational data contains no inherent nesting. Hence, an **Unnest** operator can only unnest an XML hierarchy created in the view itself. We can thus use the sound and complete view composition rules from [23] to remove **Unnest** operators in such cases. We can thus assume without loss of generality that G does not contain any **Unnest** operators.

Since all operators in G have canonical keys, the view is trigger-specifiable by Definition 4. \square

C. Event pushdown

In this part, we show our algorithm for determining what relational events can cause XML triggers to be fired.

First, we precisely define the notion of INSERT, UPDATE, and DELETE events on an operator o , given a database transition $D \xrightarrow{*} D'$:

- **INSERT**(o): There exists $t \in R(o, D')$ such that $\neg \exists t'(t' \in R(o, D) \wedge ckv_o(t) = ckv_o(t'))$.
- **DELETE**(o): There exists $t \in R(o, D)$ such that $\neg \exists t'(t' \in R(o, D') \wedge ckv_o(t) = ckv_o(t'))$.
- **UPDATE**(o, C): There exists $t \in R(o, D)$ such that for the set of columns C , $\exists t'(t' \in R(o, D') \wedge ckv_o(t) = ckv_o(t') \wedge \pi_C(v(t)) \neq \pi_C(v(t'))$).

The procedure we use is quite straightforward: for each type of operator (**Join**, **GroupBy**, etc.), and for each of the three event types, there is a set of possible input events which can directly cause that output event (see Table 4). Put another way, there is a set of rules $E_I \rightarrow E_O$, where operator I is an input to operator O , such that the event E_O can

```

1: GetSrcEvents ( $o : \text{Operator}, e : \text{Event}$ ) :
2:    $S \leftarrow \{(o', e') \mid \text{determined from } (o, e) \text{ using Table 4}\}$ 
3:    $S' \leftarrow \emptyset$ 
4:   for all  $(o', e') \in S$  do
5:     if  $o'.type = \text{Table}$  then
6:        $S' \leftarrow S' \cup \{(o', e')\}$ 
7:     else
8:        $S' \leftarrow S' \cup \text{GetSrcEvents}(o', e')$ 
9:     end if
10:  end for
11:  Return  $S'$ .

```

Figure 19. GetSrcEvents: given an operator, o , and a desired event on that operator, e , returns the set of table-level events which can cause e .

occur if E_I occurs. Thus, starting at the top operator of the *Path* graph, we can determine the set of events on all of its input operators which can directly cause the *Event* specified in the trigger. Applying these rules recursively, as shown in Figure 19, we eventually reach the base **Table** operators, at which point we have the set of all base-table events I_B such that $I_B \rightarrow \text{Event}$.

D. Supported XQuery Expressions

Our implementation supports a powerful subset of XQuery in view and trigger definitions. As mentioned in Section 2.2, the trigger's *Path* is an XPath expression, while the *Condition* as well as the parameters to the *Action* function are XQuery expressions. Rather than list the full XPath and XQuery grammar we support, we will instead highlight the differences between the grammar supported by our implementation and the XQuery 1.0 specification [27].

Figure 20 shows the restrictions we place on the XQuery 1.0 grammar. The two main restrictions are on the axes supported (we do not support parent or sibling accessors), and on type expressions, which we do not support (hence the removal of `InstanceofExpr`). There are also restrictions on the XQuery functions [28] which we support; currently, we allow only arithmetic functions which have counterparts in SQL, and we do not allow user-defined XQuery functions.

E. Proof of Theorem 2

In this section, we will prove the correctness of the **CreateANGraph** algorithm (see Figure 12).

Operator type	Input (operator, key) pairs	How to derive output key (Key_O)
Select, Project	(I, Key_I)	/* Simply propagate the key of our input operator. */ $Key_O \leftarrow Key_I$
Join	$(I_1, Key_{I_1}), \dots, (I_n, Key_{I_n})$	/* New key is the concatenation of input keys. */ $Key_O \leftarrow Key_{I_1} \cup \dots \cup Key_{I_n}$
Union	$(I_1, Key_{I_1}), \dots, (I_n, Key_{I_n})$	Let $M : C_I \rightarrow C_O$ be the mapping from the columns of input operators to columns of O . $Key_O \leftarrow \bigcup_{I_i \in I} \left(\bigcup_{c \in Key_{I_i}} M(c) \right)$
GroupBy	(I, Key_I)	$Key_O \leftarrow$ the grouping columns of O .
Table	—	$Key_O \leftarrow$ the primary key of O .

Table 3. Deriving canonical keys for XQGM operators.

Operator type	Output event	Input (operator,event) pairs
Select, Project	DELETE(O)	DELETE(I) (I is the input operator); UPDATE(I, C_σ) where C_σ are the columns used in the selection condition
	INSERT(O)	INSERT(I); UPDATE(I, C_σ)
	UPDATE(O, C)	UPDATE(I, C)
Join	DELETE(O)	DELETE(I) for any input operator I ; UPDATE(I, C), where C_I are the columns of operator I .
	INSERT(O)	INSERT(I) for any input operator I ; UPDATE(I, C)
	UPDATE(O, C)	UPDATE(I, C_I)
GroupBy	DELETE(O)	DELETE(I); UPDATE(I, G), where G is the set of grouping columns
	INSERT(O)	INSERT(I); UPDATE(I, G)
	UPDATE(O, C)	UPDATE(I, C); INSERT(I) unless $C \subseteq G$; DELETE(I) unless $C \subseteq G$
Union	DELETE(O)	DELETE(I) for any input operator I ; UPDATE(I, I_C) for any input operator I (Note that DELETE(O) could be caused by an UPDATE where a previously unique tuple becomes a duplicate.)
	INSERT(O)	INSERT(I) for any input operator I ; UPDATE(I, I_C) for any input operator I (analogously to DELETE(O))
	UPDATE(O, C)	UPDATE(I, I_C) for any input operator I

Table 4. Operator-specific rules used in event pushdown.

[31] $\langle \text{MainModule} \rangle$	\longrightarrow	$\langle \text{Prolog} \rangle \langle \text{QueryBody} \rangle$
$\langle \text{MainModule}^* \rangle$	\longrightarrow	$\langle \text{QueryBody} \rangle$
[72] $\langle \text{AxisStep} \rangle$	\longrightarrow	$(\langle \text{ForwardStep} \rangle \mid \langle \text{ReverseStep} \rangle) \mid \langle \text{Predicates} \rangle$
$\langle \text{AxisStep}^* \rangle$	\longrightarrow	$\langle \text{ForwardStep} \rangle \langle \text{Predicates} \rangle$
[89] $\langle \text{ForwardAxis} \rangle$	\longrightarrow	$(\text{child} ::)$ $(\text{descendant} ::)$ $(\text{attribute} ::)$ $(\text{self} ::)$ $(\text{descendant-or-self} ::)$ $(\text{following-sibling} ::)$ $(\text{following} ::)$
$\langle \text{ForwardAxis}^* \rangle$	\longrightarrow	$(\text{child} ::)$ $(\text{descendant} ::)$ $(\text{attribute} ::)$ $(\text{self} ::)$ $(\text{descendant-or-self} ::)$
[56] $\langle \text{AndExpr} \rangle$	\longrightarrow	$\langle \text{InstanceofExpr} \rangle$ $\{ \text{and} (\langle \text{InstanceofExpr} \rangle) \}$
$\langle \text{AndExpr}^* \rangle$	\longrightarrow	$\langle \text{ComparisonExpr} \rangle$ $\{ \text{and} (\langle \text{ComparisonExpr} \rangle) \}$

Figure 20. Supported XQuery grammar. Non-terminals marked with an asterisk (*) indicate our implementation, while the bracketed number refers to the rule in the XQuery specification [27].

E.1. Avoiding Spurious UPDATE Events

The goal of **CreateANGraph** is to produce an XQGM graph which, when evaluated, will produce the values of `OLD_NODE` and `NEW_NODE` for each XML node which was inserted, updated, or deleted. If, as we shall prove, **Create-AKGraph** (Figure 8) is correct, then the anti-join at the top of $G_{affected}$ (lines 13 and 15 of **CreateANGraph**) prevents INSERT and DELETE triggers from producing spurious output.

However, when the XML trigger is on an UPDATE event, there are cases in which spurious updates might be detected. To see why, suppose the view for our running example (Figure 3) were changed so that instead of producing the list of vendors for each product, it only produced the minimum price for each. The XQGM graph for this modified view is shown in Figure 21. Note that the only difference from the original (Figure 5) is in the **Project** and **GroupBy** (boxes 4 and 5).

Now suppose *vendor* initially contains the following two tuples:

vendor		
vid	pid	price
Amazon	P1	100.00
Buy.com	P1	60.00

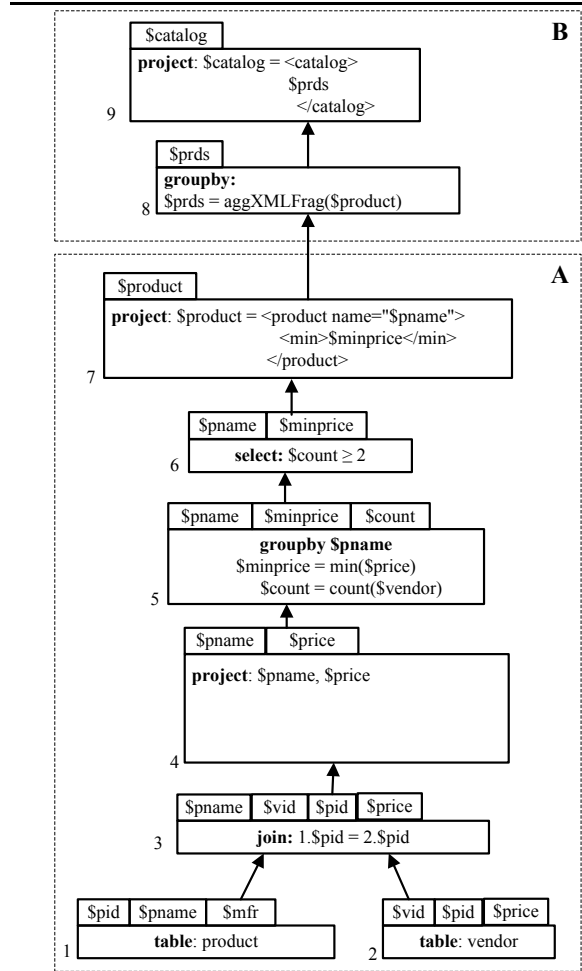


Figure 21. XQGM for modified *catalog* view.

and one vendor updates its price:

∇_{vendor}			Δ_{vendor}		
vid	pid	price	vid	pid	price
Amazon	P1	100.00	Amazon	P1	75.00

Since the view creates *product* elements, P1 will be identified by our algorithm as an affected key because Δ_{vendor} and ∇_{vendor} (potentially) affect the aggregate produced by the **GroupBy** operator in Box 5. However, since this particular update does *not* change the minimum price, the XML node for product P1 remains unchanged: both before and after the update, the node is simply:

```
<product name="P1">
  <min>60.00</min>
</product>
```

On the other hand, if the new price for *Amazon* had been, say, \$50.00, this XML node *would* have been affected. Thus, when the XML trigger is on an UPDATE event, we need to ensure that the node in question was *actually* updated. The simplest, but far from

the most efficient, solution is to place a selection condition at the top of $G_{affected}$ (Line 11 in Figure 12) which filters out those (OLD_NODE, NEW_NODE) pairs where OLD_NODE = NEW_NODE. This is implemented as a string comparison in the tagger (since it's a comparison of the full XML nodes), which has two drawbacks. First, it is expensive when the nodes are large. Second, and more importantly, it *requires* passing the entire (OLD_NODE, NEW_NODE) pair to the middleware, which prevents many of the optimizations which would otherwise be performed by XQGM graph rewrite rules when the *Condition* and *Action* do not require the entire nodes. We therefore present some optimizations and prove their correctness in Appendix F. For this section, however, we assume the use of the simpler, less-efficient approach.

E.2. Proof of Correctness of CreateAKGraph

Central to our proof of Theorem 2 is the correctness of the affected-keys algorithm, **CreateAKGraph** (Figure 8).

First, we formally define some terminology. In the following, we use $R(T, D)$ to denote the contents of table T in database state D . (In other words, $R(T, D) = R(o, D)$ where o is the XQGM operator **Table**(T).)

Definition 5 (Valid transitional tables). *For any given single-table database transition $D \xrightarrow{T} D'$, $(\nabla T, \Delta T)$ is a valid pair of transitional tables iff*

$$\begin{aligned} \nabla T &\subseteq R(T, D), \Delta T \subseteq R(T, D'), \\ \nabla T &\supseteq \{x \mid x \in R(T, D) \wedge x \notin R(T, D')\}, \\ \Delta T &\supseteq \{x \mid x \in R(T, D') \wedge x \notin R(T, D)\}, \\ \text{and } (R(T, D) - \nabla T) &= R(T, D') - \Delta T. \end{aligned}$$

Definition 6 (Hypothetical state). *For a given database state D and a transitional table dT , the hypothetical state D_{-dT} is the database state such that $R(T, D_{-dT}) = R(T, D) - dT$ and for all tables $T' \neq T$, $R(T', D_{-dT}) = R(T', D)$.*

We refer to a database transition as *monotonic* if $\nabla T = \emptyset$ or $\Delta T = \emptyset$. It follows from Definitions 5 and 6 that both $D \xrightarrow{T} D'_{-\Delta T}$ and $D'_{-\Delta T} \xrightarrow{T} D'$ are monotonic transitions.

Definition 7 (“Affected”). *A tuple t is said to be affected in view G by relational transition $D \xrightarrow{T} D'$ iff t is updated, inserted, or deleted in G by $D \xrightarrow{T} D'$ (as per Definitions 2 and 3).*

We now prove the correctness of **CreateAKGraph**.

Lemma 1 (Correctness of CreateAKGraph). *Given a view graph G , a relational table T , and a monotonic database transition $D_1 \xrightarrow{T} D_2$ with non-empty transition table dT , let $O' = \text{CreateAKGraph}(o_G, T, dT)$.*

Then $ckv_{O'}(x) \in R(O', D_2)$ for all tuples x where x is affected in G by $D_1 \xrightarrow{T} D_2$.

Proof. We prove Lemma 1 by induction on the depth of G .

Base case: depth = 1.

In this case, the view graph only consists of a single operator, **Table**(X), for some relational table X .

Suppose $T \neq X$. Since we stipulated that a database transition $D_1 \xrightarrow{T} D_2$ occurred, D_1 and D_2 are identical states except for the contents of table T . Therefore, since $R(\text{Table}(X), D_1) = R(\text{Table}(X), D_2)$, there are *no* tuples affected, so the lemma is vacuously true.

On the other hand, suppose that $T = X$. Then $\text{CreateAKGraph}(o_G, T, dT) = \pi_{T.key}(\text{Table}(dT))$. Hence, by the definition of transitional tables, $R(O', D_2)$ contains all tuples x affected by $D_1 \xrightarrow{T} D_2$.

Thus, the base case holds.

Induction Hypothesis: For a graph H of depth $\leq k$, suppose Lemma 1 holds.

We will now show that Lemma 1 holds for a graph G of depth $k + 1$. There are four cases, one for each type of operator except for **Table** (which can only occur at the leaf level of the graph).

Case 1: o_G is a GroupBy operator.

This case is handled by lines 11-18 of the algorithm.

Then there are two cases to consider, depending on the value returned by the recursive call to **CreateAKGraph**, I' . First, if $I' = \emptyset$, then there are no tuples in the input to the **GroupBy** which were affected as a result of the database transition. Since the input to the **GroupBy** is unchanged, and it merely aggregates its input, its output must also be unaffected by the transition; hence, in this case, we simply return \emptyset .

Otherwise, the algorithm creates a **Join** operator, J , joining I with I' , and then returns a new **GroupBy** operator which merely projects out the values of the grouping columns of o_G . Note that for a **GroupBy** operator, the grouping columns *are* its canonical key. By the induction hypothesis, $ckv_{I'}(x) \in R(I', D_2)$ for every tuple x affected by the transition. If $ckv_{I'}(x)$ is produced by I' , then J will produce all tuples y where $ckv_{o_G}(x) = ckv_{o_G}(y)$, and O' will produce $ckv_{O'}(y)$. In other words, for each tuple produced by operator I affected by the transition, O' produces the corresponding value of the canonical key of the **GroupBy** operator, O' .

Finally, since each tuple z produced by a **GroupBy** operator Q depends only on those input tuples w where $ckv_Q(w) = ckv_Q(z)$, the keys of all tuples affected in G by the transition $D_1 \xrightarrow{T} D_2$ are included in $R(O', D_2)$.

*Case 2: o_G is a **Select** or **Project** operator.*

This case is handled by lines 20-21 of the algorithm.

Both **Select** and **Project** take a single input, I . Suppose, by contradiction, that there exists a tuple x such that x is affected in G by $D_1 \xrightarrow{T} D_2$, but $ckv_{O'}(x) \notin R(O', D_2)$. By the algorithm (line 21), $O' = I'$, so it must also be true that $ckv_{I'}(x) \notin R(I', D_2)$. Let y be the tuple produced by I such that $ckv_I(y) = ckv_{I'}(x)$. There must be exactly one such tuple, because **Project** does not change the cardinality of its input, and **Select** can only decrease it. Therefore $ckv_{I'}(y) \notin R(I', D_2)$. By the induction hypothesis, this implies that y is *not* affected by $D_1 \xrightarrow{T} D_2$, for if it were, $ckv_{I'}(y)$ would be in $R(I', D_2)$. Both **Select** and **Project** are deterministic (given the limitations on XQuery functions laid out in Appendix D), and compute each output tuple in terms of exactly one input tuple. But this yields a contradiction, because an unaffected input tuple y resulted in an affected corresponding output tuple x .

*Case 3: o_G is a **Join**.*

This case is handled by lines 24-39 of the algorithm.

A **Join** with predicates is semantically equivalent to a **Join** with no predicates (i.e., a cross-product) followed by a **Select** imposing the predicates. Since a **Select** requires no additional operators added to the affected-keys graph (see Case 2 above), we can assume, without loss of generality, that o_G has no predicates.

Furthermore, a **Join** with fewer than 2 input operators is equivalent to a **Select**, and a **Join** with more than 2 input operators can be split up into several joins, i.e., $\text{Join}(I_1, I_2, \dots, I_n) = \text{Join}(I_1, \text{Join}(I_2, \dots, I_n))$. We therefore make the additional simplifying assumption that each **Join** has exactly 2 inputs.

The algorithm begins by invoking **CreateAKGraph** recursively on each of the inputs (I_0 and I_1). There are three possible cases to consider: first, suppose both invocations return \emptyset . By the induction hypothesis, this implies that the input to o_G is unchanged; since **Join** is deterministic, its output must also be unchanged.

The second possibility is that **CreateAKGraph** returned \emptyset for exactly one of I_0 or I_1 , and returned some I' for the other. The proof for this case is almost identical to the proof-by-contradiction used in Case 2 for **Select**, so we just sketch it out: assume that there exists a tuple x such that x is affected in G by $D_1 \xrightarrow{T} D_2$, but $ckv_{O'}(x) \notin R(O', D_2)$; then a contradiction arises because only one leg of the **Join** changed, and $I' = O'$, so the corresponding tuple $y \in R(I', D_2)$ should have been identified by I' .

The third and final possibility is that **CreateAKGraph** returned I'_0 and I'_1 for I_0 and I_1 , respectively. In this case, we produce O' by creating two **Join** operators— J_a computing the cross-product ($I'_0 \times I'_1$), and J_b computing ($I_0 \times I'_1$)—and taking their **Union**. Suppose by contradic-

tion that there exists a tuple x such that x is affected in G by $D_1 \xrightarrow{T} D_2$, but $ckv_{O'}(x) \notin R(O', D_2)$. Then, by definition of a cross-product, there exists exactly one pair of tuples (y, z) such that $x = y \cdot z$, i.e. $y \in R(I_0, D_2)$, $z \in R(I_1, D_2)$, $ckv_{I_0}(y) = ckv_{I_0}(x)$, and $ckv_{I_1}(z) = ckv_{I_1}(x)$. Since x was affected, and **Join** is deterministic, it must be the case that either y or z was also affected. Without loss of generality, assume it was y . Then, by the induction hypothesis, it must be the case that $ckv_{I'_0}(y) \in R(I'_0, D_2)$. Therefore, since $z \in R(I_1, D_2)$, there is a tuple $x' \in R(J_a, D_2)$ such that $ckv_{I'_0}(x') = ckv_{I'_0}(y)$ and $ckv_{I'_1}(x') = ckv_{I'_1}(z)$. But this yields a contradiction, since the **Union** will simply propagate x' , and $x' = ckv_{O'}(x)$, contradicting our original assumption that $ckv_{O'}(x) \notin R(O', D_2)$.

*Case 4: o_G is a **Union**.*

The final possibility is that o_G is a **Union** operator; this is handled in lines 43-53. The algorithm for **Union** calls **CreateAKGraph** recursively on each of its inputs i , and then creates a new **Union** operator computing the union of each of these results i' .

We prove the correctness for this final case by contradiction. Suppose there exists some tuple x such that x is affected in G by the transition $D_1 \xrightarrow{T} D_2$, but $x \notin R(O', D_2)$. By definition of the **Union** operator, there must be at least one input operator I such that x was affected in I . By the induction hypothesis, $ckv_{I'}(x) \in R(I', D_2)$. However, this yields a contradiction because I' is one of the inputs to O' , and the semantics of **Union** require that $\forall y(y \in I' \rightarrow y \in O')$. \square

E.3. Proof of Correctness of CreateANGraph

Before we can prove the correctness of **CreateANGraph**, we must first prove the following lemma:

Lemma 2. *For a database transition $D \xrightarrow{T} D'$ with transitional tables ΔT and ∇T , and a view graph G , if a tuple t is affected in G by $D \xrightarrow{T} D'$, then either t is affected by $D \xrightarrow{T} D'_{-\Delta T}$, or t is affected by $D'_{-\Delta T} \xrightarrow{T} D'$.*

Proof. For conciseness of notation, let $\bar{D} = D'_{-\Delta T}$. Suppose by contradiction that (1) t is affected by $D \xrightarrow{T} D'$, but (2) t is not affected by $D \xrightarrow{T} \bar{D}$ and t is not affected by $\bar{D} \xrightarrow{T} D'$. By Definitions 2 and 3, it follows from (2) that one of two cases is possible: either (3) $(t \in R(o_G, D) \wedge t \in R(o_G, \bar{D}) \wedge t \in R(o_G, D'))$ or (4) $(t \notin R(o_G, D) \wedge t \notin R(o_G, \bar{D}) \wedge t \notin R(o_G, D'))$.

If (3) holds, then $t \in R(o_G, D) \wedge t \in R(o_G, D')$, so it follows from Definitions 2 and 3 that t is neither inserted, deleted, or updated in G by $D \xrightarrow{T} D'$, contradicting our assumption (1) that t is affected by $D \xrightarrow{T} D'$.

On the other hand, if (4) holds, then $t \notin R(o_G, D) \wedge t \notin R(o_G, D')$; again, it follows from Definitions 2 and 3 that t is neither inserted, deleted, or updated in G by $D \xrightarrow{T} D'$, contradicting assumption (1). \square

We now proceed with the proof of Theorem 2, the correctness of **CreateANGraph**. Please refer to Figure 12 for the text of the algorithm, which is referenced throughout this proof.

Theorem 2. *Given an event E , view graph G , and table T , **CreateANGraph**(E, G, T) produces graph $G_{affected}$ such that for all valid database transitions $D \xrightarrow{T} D'$, $(OLD_NODE, NEW_NODE) \in R(o_{G_{affected}}, D')$ iff:*

- (a) $E = \text{UPDATE} \wedge OLD_NODE \in R(o_G, D) \wedge NEW_NODE \in R(o_G, D') \wedge ckv_{o_G}(OLD_NODE) = ckv_{o_G}(NEW_NODE) \wedge v(OLD_NODE) \neq v(NEW_NODE)$, or
- (b) $E = \text{INSERT} \wedge OLD_NODE = \emptyset \wedge NEW_NODE \in R(o_G, D') \wedge \nexists x | (x \in R(o_G, D) \wedge ckv_{o_G}(NEW_NODE) = ckv_{o_G}(x))$, or
- (c) $E = \text{DELETE} \wedge NEW_NODE = \emptyset \wedge OLD_NODE \in R(o_G, D) \wedge \nexists x | (x \in R(o_G, D') \wedge ckv_{o_G}(OLD_NODE) = ckv_{o_G}(x))$.

Proof. Lemma 1 proved that if some tuple x is affected in G by the database transition $D'_{-\Delta T} \xrightarrow{T} D'$, then $ckv_{o_G}(x) \in R(\text{CreateAKGraph}(o_G, T, \Delta T), D')$, and that if some tuple x is affected in G by $D \xrightarrow{T} D'_{-\Delta T}$, then $ckv_{o_G}(x) \in R(\text{CreateAKGraph}(o_{G_{old}}, T_{old}, \nabla T), D')$. As Lemma 2 showed, for any tuple x affected in G by $D \xrightarrow{T} D'$, it must be affected either by $D'_{-\Delta T} \xrightarrow{T} D'$ or by $D \xrightarrow{T} D'_{-\Delta T}$. Therefore **Union** created in line 6 produces a superset of affected keys. I.e., for any affected tuple x , it must be the case that $x \in R(O_u, D')$ (see line 6).

We now consider each of the three event types separately, and we prove the theorem in both directions for each.

(a) $E = \text{UPDATE}$.

Suppose $E = \text{UPDATE}$, and there exist OLD_NODE and NEW_NODE such that $OLD_NODE \in R(o_G, D) \wedge NEW_NODE \in R(o_G, D') \wedge ckv_{o_G}(OLD_NODE) = ckv_{o_G}(NEW_NODE) \wedge v(OLD_NODE) \neq v(NEW_NODE)$. This is the definition of an UPDATE event on o_G , and since we have shown that $R(O_u, D')$ contains $ckv_{o_G}(x)$ for all tuples x affected by the database transition, we can infer that $R(O_u, D')$ contains $ckv_{o_G}(OLD_NODE)$. Then, since $OLD_NODE \in R(o_G, D)$, it follows that $OLD_NODE \in R(O_{old}, D')$ (line 8); similarly, since $NEW_NODE \in R(o_G, D')$, therefore $NEW_NODE \in R(O_{new}, D')$ (line 8). Next, since there is exactly one tuple in each of $R(O_{old}, D')$ and

$R(O_{new}, D')$, the inner join (line 10) produces exactly (OLD_NODE, NEW_NODE) . Since we initially stipulated that $OLD_NODE \neq NEW_NODE$, the final **Select** (line 11) does not remove this pair from the output. Therefore, $G_{affected}$ produces the tuple (OLD_NODE, NEW_NODE) .

Conversely, suppose $E = \text{UPDATE}$, and there exists $(OLD_NODE, NEW_NODE) \in R(O_{G_{affected}}, D')$. Then, this must have been returned in line 11, so we can infer that $OLD_NODE \neq NEW_NODE$. Furthermore, OLD_NODE and NEW_NODE come from O_{old} and O_{new} , respectively, joined on their respective keys; therefore, since the key of both O_{old} (line 8) and O_{new} (line 7) is the same as the key of G , we have that $ckv_{o_G}(OLD_NODE) = ckv_{o_G}(NEW_NODE)$. Finally, since $R(O_{new}, D') \subseteq R(G, D')$, we can conclude that $NEW_NODE \in R(G, D')$. We can similarly conclude that $OLD_NODE \in R(G, D)$ because $R(O_{old}, D') \subseteq R(G, D)$.

(b) $E = \text{INSERT}$.

Next, suppose that $E = \text{INSERT}$, and there exist OLD_NODE and NEW_NODE such that $OLD_NODE = \emptyset \wedge NEW_NODE \in R(o_G, D') \wedge \nexists x | (x \in R(o_G, D) \wedge ckv_{o_G}(NEW_NODE) = ckv_{o_G}(x))$. This is the definition of an INSERT event on o_G , and since we have shown that $R(O_u, D')$ contains $ckv_{o_G}(x)$ for all tuples x affected by the database transition, we can infer that $R(O_u, D')$ contains $ckv_{o_G}(NEW_NODE)$. Then, since $NEW_NODE \in R(o_G, D')$, it follows that $NEW_NODE \in R(O_{new}, D')$; similarly, since there is no corresponding tuple x in $R(o_G, D)$, it follows that there is no tuple $y \in R(O_{old}, D')$ such that $ckv_{o_G}(y) = ckv_{o_G}(NEW_NODE)$. Therefore, the **LeftAntiJoin** (line 13) will produce (\emptyset, NEW_NODE) .

Conversely, suppose $E = \text{INSERT}$, and there exists $(OLD_NODE, NEW_NODE) \in R(O_{G_{affected}}, D')$. Then, this must have been returned in line 13, so we can infer that $OLD_NODE = \emptyset$. Furthermore, OLD_NODE and NEW_NODE come from O_{old} and O_{new} , respectively, anti-joined on their respective keys. Therefore, since the key of both O_{old} (line 8) and O_{new} (line 7) is the same as the key of G , we have that $NEW_NODE \in R(G, D')$, and $\nexists y | (y \in R(o_{O_{old}}, D') \wedge ckv_{o_G}(y) = ckv_{o_G}(NEW_NODE))$. Because $NEW_NODE \in R(O_{new}, D')$, we conclude that $ckv_{o_G}(NEW_NODE) \in R(O_u, D')$. Therefore, since we have shown that O_{old} does not contain a corresponding tuple y , it follows that $\nexists x | (x \in R(o_G, D) \wedge ckv_{o_G}(NEW_NODE) = ckv_{o_G}(x))$.

(c) $E = \text{DELETE}$.

The final case is analogous to (b). Suppose that $E = \text{DELETE}$, and there exist OLD_NODE and NEW_NODE such that $NEW_NODE = \emptyset \wedge OLD_NODE \in R(o_G, D) \wedge \nexists x | (x \in R(o_G, D') \wedge ckv_{o_G}(OLD_NODE) = ckv_{o_G}(x))$. This is the definition of a DELETE event on o_G , and since we have shown that $R(O_u, D')$ contains $ckv_{o_G}(x)$ for all tuples x affected by the database transition, we

can infer that $R(O_u, D')$ contains $ckv_{o_G}(\text{OLD_NODE})$. Then, since $\text{OLD_NODE} \in R(o_G, D)$, it follows that $\text{OLD_NODE} \in R(O_{old}, D')$; similarly, since there is no corresponding tuple x in $R(o_G, D')$, it follows that there is no tuple $y \in R(O_{new}, D')$ such that $ckv_{o_G}(y) = ckv_{o_G}(\text{OLD_NODE})$. Therefore, the **Right-AntiJoin** (line 15) will produce $(\text{OLD_NODE}, \emptyset)$.

Conversely, suppose $E = \text{DELETE}$, and there exists $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(O_{G_{affected}}, D')$. Then, this must have been returned in line 15, so we can infer that $\text{NEW_NODE} = \emptyset$. Furthermore, OLD_NODE and NEW_NODE come from O_{old} and O_{new} , respectively, anti-joined on their respective keys. Therefore, since the key of both O_{old} (line 8) and O_{new} (line 7) is the same as the key of G , we have that $\text{OLD_NODE} \in R(G, D)$, and $\nexists y | (y \in R(o_{O_{new}}, D') \wedge ckv_{o_G}(y) = ckv_{o_G}(\text{OLD_NODE}))$. Because $\text{OLD_NODE} \in R(O_{old}, D')$, we conclude that $ckv_{o_G}(\text{OLD_NODE}) \in R(O_u, D')$. Therefore, since we have shown that O_{new} does not contain a corresponding tuple y , it follows that $\nexists x | (x \in R(o_G, D') \wedge ckv_{o_G}(\text{OLD_NODE}) = ckv_{o_G}(x))$. \square

F. Optimizations for CreateANGraph

As described in Appendix E.1, in the **CreateANGraph** algorithm (Figure 12), we initially put a **Select** operator (line 11) at the top of the affected-node graph in order to ensure that OLD_NODE and NEW_NODE actually differ; however, doing this comparison in the tagger can be expensive. In this section, we identify a general class of views for which we do not have to explicitly check whether OLD_NODE and NEW_NODE differ, while still ensuring that we do not identify spurious updates - many views, including the running example in the paper, fall into this class. For views that do not fall into this class, we present a few optimizations that can push down the check to the relational engine under certain conditions.

F.1. Definitions

As Definition 5 shows, the transitional tables provided by the relational database system are actually a *superset* of the tuples which changed. This is because an update statement such as:

```
UPDATE VENDOR
SET PRICE = 1 * PRICE
```

will result in the transitional tables containing as many rows as there are *vendor* rows, even though none of the *vendor* rows actually changed in value. Therefore, **CreateAK-Graph** would produce keys of XML nodes which were not actually updated, and these would not be identified as spurious until reaching the final **Select** operator.

This problem can be avoided by replacing all references to ΔT and ∇T in the SQL trigger with $\Delta T'$ and $\nabla T'$, respectively, where $\Delta T' = \Delta T - \nabla T$ and $\nabla T' = \nabla T - \Delta T$. Then we can refine Definition 5:

Definition 8 (Pruned transitional tables). For any given single-table database transition $D \xrightarrow{T} D'$, the pruned transition tables $\blacktriangle T$ and $\blacktriangledown T$ are:

$$\blacktriangle T = \{x | x \in R(T, D') \wedge x \notin R(T, D)\}, \text{ and}$$

$$\blacktriangledown T = \{x | x \in R(T, D) \wedge x \notin R(T, D')\}.$$

For a large class of views, including the running example in the paper, we can actually remove the selection condition $\text{OLD_NODE} \neq \text{NEW_NODE}$ from **CreateANGraph** if we prune the transition tables. This class of views, which we call *injective*, has the property that there is a one-to-one mapping between each XML node produced by o_G (the top operator of the view graph) and the set of relational tuples used to construct the node.

In order to prove this claim, we must first define the notion of an injective view more formally. We begin by defining the *contributing set* of a tuple: intuitively, for any tuple t produced by an operator o , there is a set of tuples produced by each of its input operators o_i which contributes to t . For example, the contributing set of a tuple t produced by a **GroupBy** operator is the set of all input tuples having the same grouping-column value as t . For a **Project** or **Select**, the contributing set of a tuple t is the input tuple from which t is computed by projection or selection, respectively. We now formalize this notion of a contributing set of a tuple for arbitrary operators.

Definition 9 (Contributing set). Given an operator (o), one of its input operators (o_i), a database state (D), and a tuple ($t_o \in R(o, D)$), the contributing set of t_o is:

$$\begin{aligned} \zeta(t_o, o_i, o, D) = & \{t_i \in R(o_i, D) | \\ & \forall D' (t_i \in R(o_i, D') \rightarrow \\ & \exists t'_o (t'_o \in R(o, D') \wedge ckv_o(t_o) = ckv_o(t'_o)))\}. \end{aligned}$$

In the following, $\tilde{C}(o)$ is the set of columns produced by operator o . For a tuple t produced by operator o , we denote the value of columns C (where $C \subseteq \tilde{C}(o)$) as $\pi_C(t)$. We similarly denote projection for a set of tuples: $\pi_C(S) = \{\pi_C(t) | t \in S\}$. Finally, when C is a set of columns belonging to (potentially) multiple operators, then $(C|o)$ denotes the subset of C belonging to operator o ; i.e. $(C|o) = C \cap \tilde{C}(o)$.

We now define *injection* for a single operator in terms of the contributing set of each tuple it produces:

Definition 10 (Injection for operators). Given an XQGM operator (o) with a set of input operators (I), a set of o 's columns (C_o), and a subset of I 's columns (C_I): the columns C_o are injective with respect to the columns C_I (denoted as $C_I \mapsto C_o$) iff:

$$\begin{aligned} & \forall t_1, D_1, t_2, D_2 (\\ & (t_1 \in R(o, D_1) \wedge t_2 \in R(o, D_2) \wedge \pi_{C_o}(t_1) = \pi_{C_o}(t_2)) \\ & \rightarrow (\forall o_i \in I(\pi_{(C_I|o_i)}(\zeta(t_1, o_i, o, D_1))) = \\ & \pi_{(C_I|o_i)}(\zeta(t_2, o_i, o, D_2))))). \end{aligned}$$

In other words, if $C_I \mapsto C_o$, then there is a one-to-one mapping such that for each tuple t produced by that operator, $\pi_{C_o}(t)$ (the value of columns C_o in tuple t) maps to a unique set of C_I values produced by the input operator(s) I .

Definition 11 (Transitive injection). An operator o is transitively injective for C_o with respect to a table T (denoted $T \mapsto^* C_o$) iff one of the following holds:

- $o = \text{Table}(T)$ and C_o is all of T 's columns, or
- $\exists C_I((C_I \mapsto C_o) \wedge \forall o_i((o_i \in I) \rightarrow (T \mapsto^* (C_I|o_i))))$.

That is, an operator is transitively injective for a subset of its output columns, C_o , if and only if there is a one-to-one mapping such that for every tuple t produced by the operator $\text{Table}(T)$, $v_{C_o}(t)$ maps to a unique set of tuples in $\text{Table}(T)$.

Finally, we say that a view with graph G is injective for C with respect to table T if its top operator, o_G , is transitively injective for C with respect to T . Although these conditions may seem restrictive, most XML views of relational data are injective with respect to each of their base tables. For example, the original *catalog* view (Figure 5) is injective with respect to both *product* and *vendor*.

In Section F.3, we will prove the following theorem, stating that for injective views, **CreateANGraph** will not produce spurious updates if we remove the final selection condition in line 11; we refer to this modified version as **CreateANOpt**.

Theorem 3. Given an UPDATE event, a view graph G which is injective for all columns of its top operator, and table T , let $G_{\text{affected}} = \text{CreateANGraph}(\text{UPDATE}, G, T)$, and $G_{\text{opt}} = \text{CreateANOpt}(\text{UPDATE}, G, T)$. Then for all valid database transitions $D \xrightarrow{T} D'$ with pruned transition tables $\blacktriangle T$ and $\blacktriangledown T$, $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{\text{opt}}}, D')$ if and only if $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{\text{affected}}}, D')$.

F.2. Sufficient Conditions for Injection

For each operator o in a graph G , and a set of columns C_o , we can determine whether o is injective for $C_i \mapsto C_o$, based on the type of operator:

- **Project, Select, and Join.** o is injective for $C_I \mapsto C_o$ if, for input operator(s) I , there exists C_I such that for each $c_i \in C_I$, one of the following holds:
 - $c_i \in C_o$, or
 - If o is **Project** or **Select**: $\exists c \in C_o$ such that c is produced by an injective function which takes

c_i as a parameter. The most commonly-used such function is the XML constructor function.

- **GroupBy.** o is injective for $C_i \mapsto C_o$ if, for its input operator i , there exists C_i such that for each $c_i \in C_i$, one of the following holds:
 - $c_i \in C_o$, or
 - $\exists c \in C_o$ such that $c = \text{aggXMLFrag}(c_i)$.

It is easy to see that the view in Figure 5 satisfies the above conditions. Note that the above conditions are sufficient but not necessary for injection.

F.3. Correctness of Theorem 3

Given an injective view, can now prove a stronger version of Lemma 1:

Lemma 3. Given a relational table T , a view graph G which is injective for columns C w.r.t. T , and a monotonic database transition $D_1 \xrightarrow{T} D_2$ with pruned nonempty transitional table dT , let $O' = \text{CreateAKGraph}(o_G, T, dT)$. Then $\nexists x, y, z$ ($x \in R(O', D_2) \wedge y \in R(o_G, D_1) \wedge \text{ckv}_{o_G}(y) = x \wedge v_C(y) = v_C(z) \wedge z \in R(o_G, D_2) \wedge \text{ckv}_{o_G}(z) = x$).

Proof. We prove Lemma 3 by induction on the depth of G .

Base case: depth = 1.

In this case, the view graph only consists of a single operator, $\text{Table}(X)$, for some relational table X .

Suppose $X \neq T$. Then **CreateAKGraph** returns \emptyset , so there are no tuples $x \in R(O', D_2)$; the lemma is vacuously true.

Otherwise, $X = T$. Then **CreateAKGraph**(o_G, T, dT) returns $\pi_{T.\text{key}}(\text{Table}(dT))$. Suppose, by contradiction, that there exist x, y, z contradicting Lemma 3. By the definition of transitive injection, C must be the set of all columns of T ; therefore, since $v_C(y) = v_C(z)$, we infer that $y = z$. The pruned transitional table dT is either $\blacktriangle T$ or $\blacktriangledown T$. By Definition 8, if $dT = \blacktriangle T$, then $z \in R(T, D_2)$ implies that $z \notin R(T, D_1)$; otherwise $dT = \blacktriangledown T$ and $y \in R(T, D_1)$ implies that $y \notin R(T, D_2)$. In both cases, a contradiction is reached because we had concluded that $y = z$.

Thus, the base case holds.

Induction Hypothesis: For a graph H of depth $\leq k$, suppose Lemma 3 holds.

We will now show that Lemma 3 holds for a graph G of depth $k + 1$. There are four cases, one for each type of operator except for **Table** (which can only occur at the leaf level of the graph).

Case 1: o_G is a GroupBy operator.

Suppose, by contradiction, that there exist tuples x, y, z contradicting Lemma 3. We know that o_G is injective for

$C_I \mapsto C$, where C_I is some set of columns of the input operator, I . By the definition of injection, we know that the set of I -tuples used for computing $v_C(y)$ and $v_C(z)$ did not change in the transition: $\pi_{C_I}(\zeta(y, I, o_G, D_1)) = \pi_{C_I}(\zeta(z, I, o_G, D_2))$. Let $Z = \zeta(z, I, o_G, D_2)$ and $X_I = \{ckv_I(z) | z \in Z\}$. Finally, let C_g be the grouping columns of o_G (note that the set of grouping columns defines the key of a **GroupBy** operator).

By the induction hypothesis, when **CreateAKGraph** is invoked recursively (line 11), $R(I', D_2)$ will not contain any $x_I \in X_I$. Therefore, the **Join** created in line 15 will not propagate any $z \in Z$. Furthermore, there can't be a tuple $z' \notin \zeta(z, I, o_G, D_2)$ such that $\pi_{C_g}(z') = x$, because $x = \pi_{C_g}(z)$ and therefore $\pi_{C_g}(z')$ would have to be in the contributing set Z (by Definition 9). As a result, since the **Join** is not propagating any tuples t such that $\pi_{C_g}(t) = x$, the **GroupBy** created in line 17 will not produce x . This yields a contradiction, however, since our original assumption was that $x \in R(O', D_2)$.

*Case 2: o_G is a **Select** or **Project** operator.*

Suppose, by contradiction, that there exist tuples x, y, z contradicting Lemma 3. We know that o_G is injective for $C_I \mapsto C$, where C_I is some set of columns of the input operator, I . The canonical key for **Select** and **Project** is defined to be the same as the key of its input, so there is exactly one pair of input tuples y', z' such that $y' \in R(I, D_1)$, $z' \in R(I, D_2)$, $ckv_I(y') = ckv_{o_G}(y)$, and $ckv_I(z') = ckv_{o_G}(z)$. By the definition of injection, we know that $v_C(y) = v_C(z)$ implies that $v_{C_I}(y') = v_{C_I}(z')$.

Therefore, by the induction hypothesis, when **CreateAKGraph** is invoked recursively (line 20), $R(I', D_2)$ will not contain x . However, this yields a contradiction, since $O' = I'$ (line 21), and our original assumption is that $x \in R(O', D_2)$.

*Case 3: o_G is a **Join**.*

If **CreateAKGraph** returned \emptyset (line 32), then there are no tuples $x \in R(O', D_2)$, so the lemma is vacuously true.

If the recursive call to **CreateAKGraph** only returned non- \emptyset for a single leg of the **Join**, then this call will simply return I' (line 34), and therefore the proof is identical to Case 2.

Thus, we only need consider the case where O' is a union of cross-products (lines 36-39). Suppose, by contradiction, that there exist tuples x, y, z contradicting Lemma 3. We know that o_G is injective for $C_I \mapsto C$, where C_I is a set of columns of the input operators I . Since the canonical key of **Join** is the concatenation of the keys of I , we infer that for each $i \in I$, there is exactly one triple of input tuples (x'_i, y'_i, z'_i) such that $y'_i \in R(i, D_1)$, $z'_i \in R(i, D_2)$, $x' = ckv_i(y') = ckv_i(y)$, and $x' = ckv_i(z') = ckv_i(z)$. By the definition of injection, we know that $v_C(y) = v_C(z)$ implies that $v_{C_i}(y'_i) = v_{C_i}(z'_i)$.

By the induction hypothesis, when **CreateAKGraph** is invoked recursively (line 25), $R(i', D_2)$ will not contain x'_i . Therefore, the cross-product $J_a = I'_0 \times I_1$ (line 37) cannot contain x , since x is the concatenation of x'_0 and x'_1 , and we have just shown that $x'_0 \notin R(I'_0, D_2)$. By the same argument, J_b (line 38) also cannot contain x , as $x'_1 \notin R(I'_1, D_2)$. Therefore, by the semantics of the **Union** operator, the union of J_a and J_b (line 39) will not produce x . Thus, we have reached a contradiction, since our original assumption was that $x \in R(O', D_2)$.

*Case 4: o_G is a **Union**.*

Suppose, by contradiction, that there exist tuples x, y, z contradicting Lemma 3. Then by the definition of the **Union** operator, $\exists I_i \in I$ such that $y \in R(I_i, D_1)$, and $\exists I_j \in I$ such that $z \in R(I_j, D_2)$. Because o_G is transitively injective for C , it follows that I_i and I_j are both transitively injective for C .

If it were the case that $z \in R(I_i, D_2)$, this would contradict the induction hypothesis for I_i . Therefore, it must be the case that $z \notin R(I_i, D_2)$. Because we had stipulated that o_G is injective for $C_I \mapsto C$, and $y \in R(o_G, D_1) \wedge z \in R(o_G, D_2)$, it follows from Definition 10 that $\pi_C(\zeta(y, I_i, o_G, D_1)) = \pi_C(\zeta(z, I_i, o_G, D_2))$. Because $ckv_{o_G}(y) = ckv_{o_G}(z)$, therefore $\pi_C(\zeta(y, I_i, o_G, D_1)) = \pi_C(\zeta(z, I_i, o_G, D_1))$. However, this yields a contradiction because $z \in \pi_C(\zeta(z, I_i, o_G, D_1))$ but $z \notin \pi_C(\zeta(z, I_i, o_G, D_2))$. \square

We have now proven that for a single transition, **CreateAKGraph** will not produce the keys of any XML nodes whose value did not actually change. However, this is not sufficient, as **CreateAKGraph** is invoked twice (once for each partial transition). The following corollary is necessary to show that given a pair of partial transitions, **CreateAKGraph** will not produce the keys of any node that changed to an intermediate value due to $D \xrightarrow{T} D'_{-\Delta T}$, and changed back to its original value due to $D'_{-\Delta T} \xrightarrow{T} D'$.

Corollary 3.1. *Given a relational table T , a view graph G which is injective for columns C with respect to T , and a database transition $D \xrightarrow{T} D'$ with pruned transitional tables $\blacktriangle T$ and $\blacktriangledown T$:*

- (a) Let $O_{\Delta} = \text{CreateAKGraph}(o_G, T, \blacktriangle T)$. $\nexists x, y, z$ ($x \in R(O_{\Delta}, D') \wedge y \in R(o_G, D) \wedge ckv_{o_G}(y) = x \wedge v_C(y) = v_C(z) \wedge z \in R(o_G, D') \wedge ckv_{o_G}(z) = x$); and
- (b) Let $O_{\nabla} = \text{CreateAKGraph}(o_{G_{old}}, T_{old}, \blacktriangledown T)$. $\nexists x, y, z$ ($x \in R(O_{\nabla}, D') \wedge y \in R(o_G, D) \wedge ckv_{o_G}(y) = x \wedge v_C(y) = v_C(z) \wedge z \in R(o_G, D') \wedge ckv_{o_G}(z) = x$).

Proof. We prove only case (a), as (b) is analogous.

Suppose that there exist x, y, z contradicting case (a) above. We know from Lemma 3 that there must $\exists w$ such

that $w \in R(o_G, D'_{\blacktriangle T}) \wedge ckv_{o_G}(w) = x \wedge v_C(w) \neq v_C(y)$. Let the set $S = \{s \in R(T, D) | ckv_{o_G}(s) = x\}$, and similarly, $\tilde{S} = \{s \in R(T, D'_{\blacktriangle T}) | ckv_{o_G}(s) = x\}$. Using Lemma 1, it follows from $v_C(w) \neq v_C(y)$ that $S \neq \tilde{S}$. Furthermore, from the definition of injection, we infer from $v_C(y) = v_C(z)$ that $S' = \{s \in R(T, D') | ckv_{o_G}(s) = x\} = S$.

By the definition of pruned transitional tables, $\blacktriangle T \cap \blacktriangledown T = \emptyset$, so by Definition 5 it follows from $S = S'$ that $S \cap \blacktriangledown T = \emptyset$. However, this yields a contradiction: we had previously concluded that $S \neq \tilde{S}$, which would require $s \in \blacktriangledown T$ for some $s \in S$, and therefore $S \cap \blacktriangledown T \neq \emptyset$. \square

Finally, we can prove that **CreateANOpt** will not produce spurious updates.

Theorem 3. *Given an UPDATE event, a view graph G which is injective for all columns of its top operator, and table T , let $G_{affected} = \text{CreateANGraph}(\text{UPDATE}, G, T)$, and $G_{opt} = \text{CreateANOpt}(\text{UPDATE}, G, T)$. Then for all valid database transitions $D \xrightarrow{T} D'$ with pruned transition tables $\blacktriangle T$ and $\blacktriangledown T$, $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{opt}}, D')$ if and only if $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{affected}}, D')$.*

Proof. It is easy to see that the “if” direction is true: from the definition of pruned transitional tables, it follows that pruned transitional tables also satisfy the definition of transitional tables, so the correctness of Theorem 2 is not affected by pruning the transitional tables. The only difference then is that **CreateANOpt** does not perform the final selection. Since **Select** can only decrease the cardinality of its input, there will be no affected nodes lost as a result of the change. Thus, $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{affected}}, D')$ implies that $(\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{opt}}, D')$.

We prove the converse by contradiction. Suppose $\exists (\text{OLD_NODE}, \text{NEW_NODE}) \in R(o_{G_{opt}}, D')$ such that $(\text{OLD_NODE}, \text{NEW_NODE}) \notin R(o_{G_{affected}}, D')$. Since $G_{affected} = \text{Select}_{(\text{OLD_NODE} \neq \text{NEW_NODE})}(G_{opt})$, it must be the case that $\text{OLD_NODE} = \text{NEW_NODE}$. For brevity, we’ll refer to this node as n .

By line 10, it must be the case that $n \in R(O_{old}) \wedge n \in R(O_{new})$. Therefore, by the definition of O_{new} and O_{old} (lines 7 and 8), it follows that $x \in R(O_u, D')$ such that $ckv_{o_G}(n) = x$. From the definition of O_u (line 6), we see that either (a) $x \in R(O_{\Delta key})$ or (b) $x \in R(O_{\nabla key})$. However, this yields a contradiction: in case (a) this violates Corollary 3.1(a), and in case (b) it violates Corollary 3.1(b). Therefore, our initial assumption that $n \in R(o_{G_{opt}}, D')$ must have been false. \square

F.4. Additional Optimizations

In the previous section, we showed that for injective view graphs, we can avoid performing an expensive tagger-

level comparison of **OLD_NODE** and **NEW_NODE** for **UPDATE** events. If a view is not injective, then we generally need to keep this selection condition. In certain cases, however, we can still optimize the performance by pushing down the selection condition to the relational level.

If a view graph G is injective except for the presence of a non-injective aggregate functions (e.g. **min**, **max**, **count**, etc.) in **GroupBy** operators, such as in the example shown in Appendix E.1, then it is not necessary to compare the *entire* old and new XML nodes. Instead, it is only necessary to compare the values of these aggregates. Since this is a comparison of numeric values with no nesting involved, it can be pushed down to the relational engine, thus avoiding the need to perform an expensive string comparison in the tagger.

This is just one of many possible optimizations to avoid performing a tagger-level comparison for non-injective views. A possible direction of future work is to identify the general class of views where the top-level selection can be pushed down to the relational level.

G. Additional Experimental Results

In this section, we show the experimental results of varying the parameter *leaf tuples per XML element*, *number of leaf tuples*, and *number of satisfied triggers*. In all experiments, we vary the parameter of interest and use default values for the rest. Note that varying these three parameters, unlike the two parameters evaluated in Section 6, changes the number of tuples inserted into the temporary table by the SQL trigger action. This introduces additional overhead not directly relevant to trigger processing. To isolate this effect, we first compute *all* the rows produced by the trigger, but only insert the *maximum* row into the temporary table (by using the max aggregate function), thus keeping the cost of the relational inserts constant.

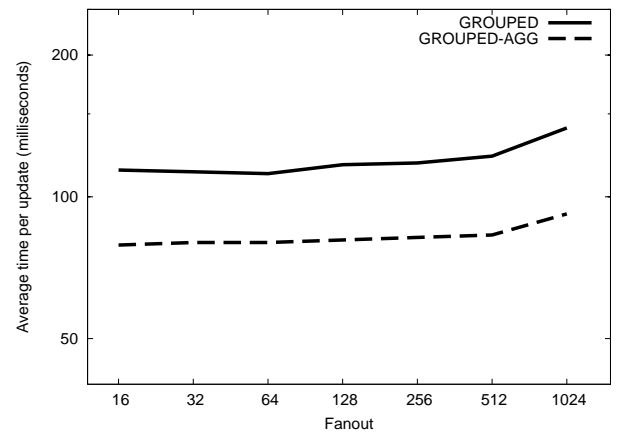


Figure 22. Varying the fanout.

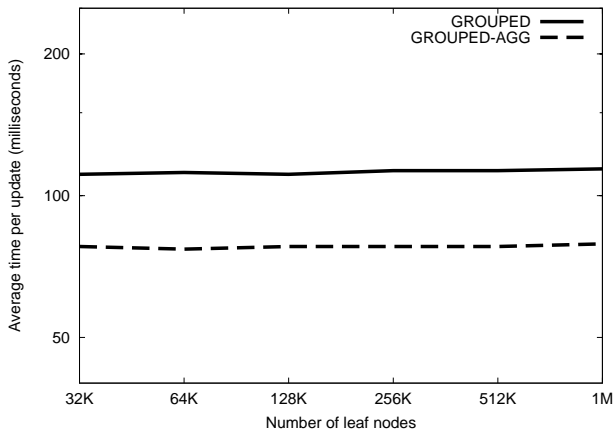


Figure 23. Varying the data size.

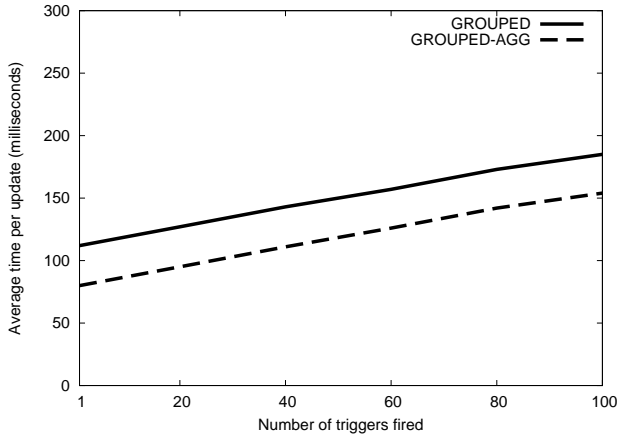


Figure 24. Varying the number of fired triggers.

G.1. Varying Leaf Tuples per XML Element

Figure 22 shows the effect of varying the *fanout*, the number of leaf tuples per XML element. GROUPED and GROUPED-AGG have the same relative performance, and there is only a small increase in runtime as the fanout increases. This increase is due primarily to the fact that the OuterUnion intermediate result grows as OLD_NODE and NEW_NODE become larger.

G.2. Varying Number of Leaf Tuples

We vary the data size by varying the number of leaf tuples; the result is shown in Figure 23. GROUPED and GROUPED-AGG scale gracefully when the data size increases. This is because, although the total number of leaf tuples increases, the number of leaf nodes in the *affected* XML element remains the same. This graph shows that our system indeed benefits from not materializing the entire XML view, so that we only need to compute a small fraction of leaf nodes.

G.3. Varying Number of Satisfied Triggers

Figure 24 shows the effect of varying the number of satisfied triggers. GROUPED and GROUPED-AGG have the same relative performance and their runtime linearly increases with the number of satisfied triggers. This is because the number of computed (OLD_NODE, NEW_NODE) pairs increases with the number of satisfied triggers.